Download tutorials.pdf (printable/viewable PDF document containing all tutorials).

Contents

Introduction

Background and Terminology

A Laplace Solver Using Simple Jacobi Iteration

Red/Black Update

Calculating Residuals

Further Topics

Pointwise Operations

Indirect Addressing

Meshes, Centerings, Geometries, and Fields

More on Meshes, Centerings, Geometries, and Fields

Particles

Particles and Fields

Text Input and Output

Object Input and Output

Compiling, Running, and Debugging POOMA Programs

Appendices

A Quick Self-Test

Managing Threads Explicitly

Recommended Reading

Legal Notice

New Material

Text I/O

Object I/O

New `Tensor` functionality

Subscriptable expressions

Component views

POOMA Tutorials
Introduction

This document is an introduction to POOMA v2.1, a C++ class library for high-performance scientific computation. POOMA runs efficiently on single-processor desktop machines, shared-memory multiprocessors, and parallel supercomputers containing dozens or hundreds of processors. What's more, by making extensive use of the advanced features of the ANSI/ISO C++ standard---particularly templates---POOMA presents a compact, easy-to-read interface to its users.

Earlier releases of POOMA v2 provided multi-dimensional arrays using a wide variety of storage schemes and parallel decompositions, multi-threading, and out-of-order execution for maximum performance. This release adds fields, coordinate systems, meshes, efficient differential operators, and particles. POOMA v2.2, which will be released in early November 1999, will support efficient distributed-memory parallelism.

To see why you might want to build your programs using POOMA, consider the following simple Laplace solver using Jacobi iteration on a fixed-size grid:

```
#include "Pooma/Arrays.h"
#include <iostream>

// The size of each side of the domain.
const int N = 20;

int
main(
    int                 argc,           // argument count
    char *              argv[]          // argument list
){
    // The array we'll be solving for
    Array<2> x(N, N);
    x = 0.0;

    // The right hand side of the equation (spike in the center)
    Array<2> b(N, N);
    b = 0.0;
    b(N/2, N/2) = -1.0;

    // Specify the interior of the domain
    Interval<1> I(1, N-2), J(1, N-2);

    // Iterate 200 times
    for (int i=0; i<200; ++i)
    {
        x(I,J) = 0.25*(x(I+1,J) + x(I-1,J) + x(I,J+1) + x(I,J-1) - b(I,J));
    }
```

```
    // Print out the result
    std::cout << x << std::endl;
}
```

The syntax is very similar to that of Fortran 90: a single assignment fills an entire array with a scalar value, subscripts express ranges as well as single points, and so on. In fact, the combination of C++ and POOMA provides so many of the features of Fortran 90 that one might well ask whether it wouldn't better to just use the latter language.

The simple answer is that the abstraction facilities of C++ are much more powerful than those in Fortran. A more powerful answer is economics. While the various flavors of Fortran are still the *lingua franca* of scientific computing, Fortran's user base is shrinking, particularly in comparison to C++. Networking, graphics, database access, and operating system interfaces are available to C++ programmers long before they're available in Fortran (if they become available at all). What's more, support tools such as debuggers and memory inspectors are primarily targeted at C++ developers, as are hundreds of books, journal articles, and web sites.

Until recently, Fortran has had two powerful arguments in its favor: legacy applications and performance. However, the importance of the former is diminishing as the invention of new algorithms force programmers to rewrite old codes, while the invention of techniques such as expression templates has made it possible for C++ programs to match, or exceed, the performance of highly optimized Fortran 77.

POOMA was designed and implemented by scientists working at the Los Alamos National Laboratory's Advanced Computing Laboratory. Between them, these scientists have written and tuned large applications on almost every commercial and experimental supercomputer built in the last two decades. As the technology used in those machines migrates down into departmental computing servers and desktop multiprocessors, POOMA is a vehicle for its designers' experience to migrate as well. In particular, POOMA's authors understand how to get good performance out of modern architectures, with their many processors and multi-level memory hierarchies, and how to handle the subtly complex problems that arise in real-world applications.

Finally, POOMA is free for non-commercial use (i.e., your tax dollars have already paid for it). You can read its source, extend it to handle platforms or problem domains that the core distribution doesn't cater to, or integrate it with other libraries and your current application, at no cost. For more information, please see the license information included in the appendix.

Of course, nothing is perfect. At the time of this release, some C++ compilers still do not support the full ANSI/ISO C++ standard. Please refer to the appendix for a list of those that do.

A second compiler-related problem is that most compilers produce very long, and very cryptic, error messages if they encounter an error while expanding templated functions and classes, particularly if those functions and classes are nested. Since POOMA uses templates extensively, it is not uncommon for a single error to result in several pages of complaints from a compiler. The appendix on error messages discusses some strategies that can be used to find the root cause of such errors. Programs that use templates extensively are also still sometimes slower to compile than programs that do not, and the executables produced by some compilers can be surprisingly large.

Finally, some debuggers still provide only limited support for inspecting templated functions and classes. All of these problems are actively being addressed by vendors, primarily in response to the growing popularity of the Standard Template Library, or STL. Once again, the large (and growing) user base for C++ means that scientific programmers can take advantage of the fact that even the best tools are constantly being improved.

The body of this tutorial starts with a discussion of the background to POOMA, including key technologies such as caching, compiler optimization, and C++ templates. The individual tutorials take a simple program---the Laplace solver shown earlier---and add more and more functionality to it, until it is able to run on multiple processors and to control its own termination by calculating user-defined residuals.

Before you start reading these tutorials, however, you may wish to take a look at the short quiz included in the appendix. POOMA does require some familiarity with some of the less well-known features of C++; if you do not feel comfortable with the questions and their answers, you may wish to have a look at one of the books in the recommended reading list before proceeding.

You may also wish to look at the POOMA web site for updates, bug fixes, and discussion of the library and how it can be used. If you have any questions about POOMA or its terms of use, or if you need help downloading or installing POOMA, please send mail to pooma@acl.lanl.gov.

[Home] [Next]

POOMA Tutorials
Background and Terminology

Contents:

Introduction
Modern Architectures
POOMA's Parallel Execution Model
Optimization
Templates
The Standard Template Library
Expression Templates

Introduction

Object-oriented programming languages like C++ make development easier, but performance tuning harder. The same abstractions that allow programmers to focus on what their program is doing, rather than how it is doing it, also make it harder for compilers to re-order operations, predict how many times a loop will be executed, or re-use an area of memory instead of making an unnecessary copy.

For example, suppose that a class `FloatVector` is being used to store and operate on vectors of floating-point values. As well as constructors, a destructor, and element access methods, this class also has overloaded operators that add, multiply, and assign whole vectors:

```
class FloatVector
{
  public :
    FloatVector();                      // default constructor

    FloatVector(                        // value constructor
        int size,                       // ..size of vector
        float val                       // ..initial element value
    );

    FloatVector(                        // copy constructor
        const FloatVector& v            // ..what to copy
    );

    virtual ~FloatVector();             // clean up

    float getAt(                        // get an element
        int index                       // ..which element to get
```

```
    ) const;

    void setAt(                          // change an element
        int index,                       // ..which element to set
        float val                        // ..new value for element
    );

    FloatVector operator+(               // add, creating a new vector
        const FloatVector& right         // ..thing being added
    );

    FloatVector operator*(               // multiply (create result)
        const FloatVector& right         // ..thing being multiplied
    );

    FloatVector& operator=(              // assign, returning target
        const FloatVector& right         // ..source
    );

  protected :
    int len_;                            // current length
    float* val_;                         // current values
};
```

Look closely at what happens when a seemingly-innocuous statement like the following is executed:

```
FloatVector V, W, X, Y;
// initialization
V = W * X + Y;
```

`W*X` creates a new `FloatVector`, and fills it with the elementwise product of `W` and `X` by looping over the raw block of `float`s encapsulated by those two vectors. The call to the addition operator then creates another temporary `FloatVector`, and executes another loop to fill it. The call to the assignment operator doesn't create a third temporary, but does execute a third loop. The net result is that our statement does the equivalent of the following code:

```
FloatVector V, W, X, Y;
// initialization

FloatVector temp_1;
for (int i=0; i<vector_size; ++i)
{
    temp_1.setAt(i, W.getAt(i) * X.getAt(i));
}

FloatVector temp_2;
for (int i=0; i<vector_size; ++i)
{
    temp_2.setAt(i, temp_1.getAt(i) + Y.getAt(i));
}

for (int i=0; i<vector_size; ++i)
```

```
{
    V.setAt(i, temp_2.getAt(i));
}
```

Clearly, if this program was written in C instead of C++, the three loops would have been combined, and the two temporary vectors eliminated, to create the more efficient code shown below:

```
FloatVector V, W, X, Y;
// initialization
for (int i=0; i<vector_size; ++i)
{
    V.setAt(i, W.getAt(i) * X.getAt(i) + Y.getAt(i));
}
```

Turning the compact C++ expression first shown into the single optimized loop shown above is beyond the capabilities of existing commercial compilers. Because operations may involve aliasing---i.e., because an expression like `V=W*X+V` can assign to a vector while also reading from it---optimizers must err on the side of caution, and neither eliminate temporaries nor fuse loops. This has led many programmers to believe that C++ is intrinsically less efficient than C or Fortran 77, and that however good object-oriented languages are for building user interfaces, they will never deliver the performance needed for modern scientific and engineering applications.

The good news is that this conclusion is wrong. By making full use of the features of the new ANSI/ISO C++ standard, the POOMA library can give a modern C++ compiler the information it needs to compile C++ programs that achieve Fortran 77 levels of performance. What's more, POOMA does not sacrifice either readability or usability in order to achieve this: in fact, POOMA programs are more portable, and more readable, than many of their peers.

In order to understand how and why POOMA does what it does, it is necessary to have at least some understanding of the architecture of a modern RISC-based computer, how compilers optimize code, and what C++ templates can and cannot do. The sections below discuss each of these topics in turn.

Modern Architectures

One of the keys to making modern RISC processors go fast is extensive use of caching. A computer uses a cache to exploit the spatial and temporal locality of most programs. The former term means that if a process accesses address *A*, the odds are good that it will access addresses near *A* shortly thereafter. The latter means that if a process accesses a value, it is likely to access that value again shortly thereafter. An example of spatial locality is access to the fields of record structures in high-level languages; an example of temporal locality is the repeated use of a loop index variable to subscript an array.

Since the extra hardware that makes a cache fast also makes it expensive, modern computer memory is organized as a set of increasingly large, but increasingly slow, layers. For example, the memory hierarchy in a 500MHz DEC 21164 Alpha typically looks like this:

Register	2ns
L1 on-chip cache	4ns
L2 on-chip cache	15ns
L3 off-chip cache	30ns
Main memory	220ns

Caches are usually built as associative memories. In a normal computer memory, a location is accessed by specifying its physical address. In an associative memory, on the other hand, each location keeps track of its

own current logical address. When the processor tries to read or write some address *A*, each cache location checks to see whether it is supposed to respond.

In practice, some restrictions are imposed. The associativity is often restricted to sets of two, four or eight, so that every cache line isn't eligible to cache every possible memory reference. The values in a cache are then usually grouped into lines containing from four to sixteen words each. Together, these bring the cost down---the cost of each piece of address-matching hardware is amortized over several cache locations---but can also lead to thrashing: if a program tries to access values at regularly-spaced intervals, it could find itself loading a cache line from memory, using just one of the values in the line, and then immediately replacing that whole line with another one. One of the key features of POOMA is that it reads and writes memory during vector and matrix operations in ways that are much less likely to lead to thrashing. While this makes the implementation of POOMA more complicated, it greatly increases its performance.

At the same time as some computer architects were making processors simpler, others were making computers themselves more complex by combining dozens or hundreds of processors in a single machine. The simplest way to do this is to just attach a few extra processors to the computer's main bus. Such a design allows a lot of pre-existing software to be recycled; in particular, since most operating systems are written so that process execution may be interleaved arbitrarily, they can often be re-targeted to multiprocessors with only minor modifications. Similarly, if a loop performs an operation on each element of an array, and the operations are independent of one another, then each of *P* processors can run *1/P* of the loop iterations independently.

The weakness of shared-bus multiprocessors is the finite bandwidth of the bus. As the number of processors increases, the time each one spends waiting to use the bus also increases. To date, this has limited the practical size of such machines to about two dozen processors.

Today's answer to this problem is to give each processor its own memory, and to use a network to connect those processor/memory nodes together. One advantage of this approach is that each node can be built using off-the-shelf hardware, such as a PC motherboard. Another advantage is that each processor's reads or writes of its own memory will be very fast. Remote reads and writes may either be done automatically by system software and hardware, or explicitly, using libraries such as PVM and MPI. So long as there is sufficient locality in programs---i.e., so long as most references are local---this scheme can deliver very high performance.

Of course, distributed-memory machines have problems too. In particular, once memory has been divided up in this way, all pointers are not created equal. On a distributed memory machine with global addressing (such as an SGI Origin 2000), dereferencing a pointer to remote memory will be considerably slower than local memory. On a distributed memory machine without global addressing (like a cluster of Linux boxes), a pointer cannot safely be passed between processes running on different processors. Similarly, if a data structure such as an array has been decomposed, and its components spread across the available processors so that each may work on a small part of it, a small change to an algorithm may have a large effect on performance. Another of POOMA's strengths is that it automatically manages data distribution to achieve high performance in a nonuniform shared memory machine. This not only lets programmers concentrate on algorithmic issues, it also saves them from having to learn the quirks of the architectures they want to run their programs on.

POOMA's Parallel Execution Model

In order to be able to cope with the variations in machine architecture noted above, POOMA's parallel execution model is defined in terms of one or more contexts, each of which may host one or more threads. A *context* is a distinct region of memory in some computer. The threads associated with the context can access data in that memory region and can run on the processors associated with that context. Threads running in different contexts cannot access memory in other contexts.

A single context may include several physical processors, or just one. Conversely, different contexts do not have to be on separate computers---for example, a 32-node SMP machine could have up to 32 separate contexts. This release of POOMA only supports a single context for each application, but can use multiple threads in the

context on supported platforms. Support for multiple contexts will be added in an upcoming release.

Optimization

Along with interpreting the footnotes in various language standards, inventing automatic ways to optimize programs is a major preoccupation of today's compiler writers. Since a program is a specification of a function mapping input values to outputs, it ought to be possible for a sufficiently clever compiler to find the sequence of instructions that would calculate that function in the least time.

In practice, the phrase "sufficiently clever" glosses over some immense difficulties. If computer memories were infinitely large, so that no location ever needed to be used more than once, the task would be easier. However, programmers writing Fortran 77 and C++ invariably save values from one calculation to use in another (i.e., perform assignments), use pointers or index vectors to access data structures, or use several different names to access a single data structure (such as segments of an array). Before applying a possible optimization, therefore, a compiler must be able to convince itself that the optimization will not have unpleasant side effects.

To make this more concrete, consider the following sequence of statements:

```
A = 5 * B + C;                  // S1
X = (Y + Z) / 2;                // S2
D = (A + 2) / E;                // S3
C = F(I, J);                    // S4
A = I + J;                      // S5
if (A < 0)                      // S6
{
    B = 0;
}
```

S_1 and S_2 are independent, since no variable appears in both. However, the result of S_3 depends on the result of S_1, since `A` appears on the right hand side of S_1. Similarly, S_4 sets the value of `C`, which S_1 uses, so S_4 must not take place before S_1 has read the previous value of `C`. S_5, which also sets the value of `A`, depends on S_1, since S_1 must not perform its assignment after S_5's if the result of the program are to be unchanged. Finally, S_6 depends on S_5 because a value set in S_5 is used to control the behavior of S_6.

While it is easy to trace the relationships in this short program by hand, it has been known since the mid-1960s that the general problem of determining dependencies among statements in the presence of conditional branches is undecidable. The good news is that if all we want are sufficient, rather than necessary, conditions---i.e. if erring on the side of caution is acceptable---then the conditions which S_i and S_j must satisfy in order to be independent are relatively simple.

This analysis becomes even simpler if we restrict our analysis to basic blocks. A basic block is a sequence of statements which can only be executed in a particular order. Basic blocks have a single entry point, a single exit point, and do not contain conditional branches or loop-backs. While they are usually short in systems programs like compilers and operating systems, most scientific programs contain basic blocks which are hundreds of instructions long. One of the aims of POOMA is to use C++ templates to make it easier for compilers to find and optimize basic blocks. In particular, as templates are expanded during compilation of POOMA programs, temporary variables that can confuse optimizers are automatically eliminated.

Another goal of POOMA's implementation is to make it easier for compilers to track data dependencies. In C++, an array is really just a pointer to the area of memory that has been allocated to store the array's values. This makes it easy for arrays to overlap and alias one another, which is often useful in improving performance, but it

also makes it very difficult for compilers to determine when two memory references are, or are not, independent.

For example, suppose we re-write the first three statements in the example above as follows (using the '*' notation of C++ to indicate a pointer de-reference):

```
*A = *B + *C;                          // T1
*X = (*Y + *Z) / 2;                    // T2
*D = (*A + 2) / *E;                    // T3
```

In the worst case, all eight pointers could point to the same location in memory, which would make this calculation equivalent to:

```
J = 2 * (J + 1) / J;
```

(where `J` is the value in that one location). At the other extreme, each pointer could point at a separate location, which would mean that the calculation would have a completely different result. Again, as templates are expanded during the compilation of POOMA programs, the compiler is automatically given the extra information it needs to discriminate between cases like these, and thereby deliver better performance.

Templates

So what exactly is a C++ template? One way to look at them is as an improvement over macros. Suppose, for example, that you wanted to create a set of classes to store pairs of `int`s, pairs of `float`s, and so on. In C, or pre-standardization versions of C++, you might first define a macro:

```
#define DECLARE_PAIR_CLASS(name_, type_)                              \
class name_                                                           \
{                                                                     \
  public :                                                            \
    name_();                           // default constructor       \
    name_(type_ left, type_ right);    // value constructor         \
    name_(const name_& right);         // copy constructor          \
    virtual ~name_();                  // destructor                \
    type_& left();                     // access left element       \
    type_& right();                    // access right element      \
                                                                      \
  protected :                                                         \
    type_ left_, right_;               // value storage             \
};
```

You could then use that macro to create each class in turn:

```
DECLARE_PAIR_CLASS(IntPair, int)
DECLARE_PAIR_CLASS(FloatPair, float)
```

A better way to do this in standard C++ is to declare a template class, and then instantiate that class when and as needed:

```
template<class DataType>
class Pair
{
  public :
```

```
    Pair();                                 // default constructor
    Pair(DataType left,                     // value constructor
         DataType right);
    Pair(const Pair<DataType>& right);      // copy constructor
    virtual ~Pair();                        // destructor
    DataType& left();                       // access left element
    DataType& right();                      // access right element

  protected :
    DataType left_, right_;                 // value storage
};
```

Here, the keyword `template` tells the compiler that the class cannot be compiled right away, since it depends on an as-yet-unknown data type. When the declarations:

```
Pair<int>   pairOfInts;
Pair<float> pairOfFloats;
```

are seen, the compiler finds the declaration of `Pair`, and instantiates it once for each underlying data type.

Templates can also be used to define functions, as in:

```
template<class DataType>
void swap(DataType& left, DataType& right)
{
    DataType tmp(left);
    left  = right;
    right = tmp;
}
```

Once again, this function can be called with two objects of any matching type, without any further work on the programmer's part:

```
int i, j;
swap(i, j);

Shape back, front;
swap(back, front);
```

Note that the implementation of `swap()` depends on the actual data type of its arguments having both a copy constructor (so that `tmp` can be initialized with the value of `left`) and an assignment operator (so that `left` and `right` can be overwritten). If the actual data type does not provide either of these, the particular instantiation of `swap()` will fail to compile.

Note also that `swap()` can be made more flexible by not requiring the two objects to have exactly the same type. The following re-definition of `swap()` will exchange the values of any two objects, provided appropriate assignment and conversion operators exist:

```
template<class LeftType, class RightType>
void swap(LeftType& left, RightType& right)
{
    LeftType tmp(left);
    left  = right;
```

```
        right = tmp;
    }
```

Finally, the word `class` appears in template definitions because any valid type, such as integers, can be used. The code below defines a template for a small fixed-size vector class, but does not fix either the size or the underlying data type:

```
template<class DataType, int FixedSize>
class FixedVector
{
  public :
    FixedVector();                      // default constructor
    FixedVector(DataType filler);       // value constructor
    virtual ~FixedVector();             // destructor

    FixedVector(                        // copy constructor
        const FixedVector<DataType, FixedSize>& right
    );

    FixedVector<DataType>&              // assignment
    operator=(
        const FixedVector<DataType, FixedSize>& right
    );

    DataType& operator[](int index);    // element access

  protected :
    DataType storage[FixedSize];        // fixed-size storage
};
```

It is at this point that the possible performance advantages of templated classes start to become apparent. Suppose that the copy constructor for this class is implemented as follows:

```
template<class DataType, int FixedSize>
FixedVector::FixedVector(
    const FixedVector<DataType, FixedSize>& right
){
    for (int i=0; i<FixedSize; ++i)
    {
        storage[i] = right.storage[i];
    }
}
```

When the compiler sees a use of the copy constructor, such as:

```
template<class DataType, int FixedSize>
void someFunction(FixedVector<DataType, FixedSize> arg)
{
    FixedVector<DataType, FixedSize> tmp(arg);
    // operations on tmp
}
```

it knows the size as well as the underlying data type of the objects being manipulated, and can therefore perform

many more optimizations than it could if the size were variable. What's more, the compiler can do this even when different calls to `someFunction()` operate on vectors of different sizes, as in:

```
FixedVector<double, 8> splineFilter;
someFunction(splineFilter);

FixedVector<double, 22> chebyshevFilter;
someFunction(chebyshevFilter);
```

Automatic instantiation of templates is both convenient and powerful, but does have one drawback. Suppose the `Pair` class shown earlier is instantiated in each of two separate source files to create a pair of `int`s. The compiler and linker could:

1. treat the two instantiations as completely separate objects;
2. detect and eliminate redundant instantiations; or
3. avoid redundancy by not instantiating templates until the program as a whole was being linked.

The first of these can lead to very large programs, as a commonly-used template class may be expanded dozens of times. The second is difficult to do, as it involves patching up compiled files as they are being linked. Most recent versions of C++ compilers are therefore taking the third approach, but POOMA users should be aware that older versions might still produce much larger executables than one would expect.

The last use of templates that is important to this discussion is member templates, which are a logical extension of templated functions. This feature was added to the ANSI/ISO C++ standard rather late, but has proved to be very powerful. Just as a templated function is instantiated on demand for different types of arguments, so too are templated methods instantiated for a class when and as they are used. For example, suppose a class is defined as follows:

```
class Example
{
  public :
    Example();                          // default constructor
    virtual ~Example();                 // destructor

    template<class T>
    void foo(T object)
    {
        // some operation on object
    }
};
```

Whenever the method `foo()` is called with an object of a particular type, the compiler instantiates the method for that type. Thus, both of the following calls in the following code are legal:

```
Example e;
Shape box;
e.foo(5);                               // instantiate for int
e.foo(box);                             // instantiate for Shape
```

The Standard Template Library

The best-known use of templates to date has been the Standard Template Library, or STL. The STL uses templates to separate containers (such as vectors and lists) from algorithms (such as finding, merging, and sorting). The two are connected through the use of *iterators*, which are classes that know how to read or write particular containers, without exposing the actual type of those containers.

For example, consider the following code fragment, which replaces the first occurrence of a particular value in a vector of floating-point numbers:

```
void replaceFirst(vector<double> & vals, double oldVal, double newVal)
{
    vector<double>::iterator loc =
        find(vals.begin(), vals.end(), oldVal);
    if (loc != vals.end())
      *loc = newVal;
}
```

The STL class `vector` declares another class called `iterator`, whose job it is to traverse a `vector`. The two methods `begin()` and `end()` return instances of `vector::iterator` marking the beginning and end of the vector. STL's `find()` function iterates from the first of its arguments to the second, looking for a value that matches the third argument. Finally, dereferencing (`operator*`) is overloaded for `vector::iterator`, so that `*loc` returns the value at the location specified by `loc`.

If we decide later to store our values in a list instead of in a vector, only the declaration of the container type needs to change, since `list` defines a nested iterator class, and `begin()` and `end()` methods, in exactly the same way as `vector`:

```
void replaceFirst(list<double> & vals, double oldVal, double newVal)
{
    list<double>::iterator loc =
        find(vals.begin(), vals.end(), oldVal);
    if (loc != vals.end())
      *loc = newVal;
}
```

If we go one step further, and use a `typedef` to label our container type, then nothing in `findValue()` needs to change at all:

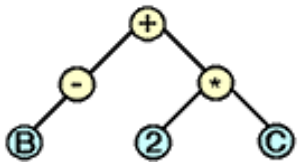
```
typedef vector<double> Storage;
// typedef list<double> Storage;

void replaceFirst(Storage<double> & vals, double oldVal, double newVal)
{
    Storage<double>::iterator loc =
        find(vals.begin(), vals.end(), oldVal);
    if (loc != vals.end())
      *loc = newVal;
}
```

The performance of this code will change as the storage mechanism changes, but that's the point: STL-based code can often be tuned using only minor, non-algorithmic changes. As the tutorials will show, POOMA borrows many ideas from the STL in order to separate interface from implementation, and thereby make

optimization easier. In particular, POOMA's arrays are actually more like iterators, in that they are an interface to data, rather than the data itself. This allows programmers to switch between dense and sparse, or centralized and distributed, array storage, with only minor, localized changes to the text of their programs.

Expression Templates

Parse trees are commonly used by compilers to store the essential features of the source of a program. The leaf nodes of a parse tree consist of atomic symbols in the language, such as variable names or numerical constants. The parse tree's intermediate nodes represent ways of combining those values, such as arithmetic operators and `while` loops. For example, the expression `-B+2*C` could be represented by the parse tree:



Parse trees are often represented textually using prefix notation, in which the non-terminal combiner and its arguments are strung together in a parenthesized list. For example, the expression `-B+2*C` can be represented as `(+ (-B)(* 2 C))`.

What makes all of this relevant to high-performance computing is that the expression `(+ (-B)(* 2 C))` could equally easily be written `BinaryOp<Add, UnaryOp<Minus, B>, BinaryOp<Multiply, Scalar<2>, C>>`: it's just a different notation. However, this notation is very similar to the syntax of C++ templates --- so similar, in fact, that it can actually be implemented given a careful enough set of template definitions. As discussed earlier, by providing more information to the optimizer as programs are being compiled, template libraries can increase the scope for performance optimization.

Any facility for representing expressions as trees must provide:

- a representation for leaf nodes (operands);
- a way to represent operations to be performed at the leaves (i.e. functions on individual operands);
- a representation for non-leaf nodes (operators);
- a way to represent operations to be performed at non-leaf nodes (i.e. combiners);
- a way to pass information (such as the function to be performed at the leaves) downward in the tree; and
- a way to collect and combine information moving up the tree.

C++ templates were not designed with these requirements in mind, but it turns out that they can satisfy them. The central idea is to use the compiler's representation of type information in an instantiated template to store operands and operators. For example, suppose that a set of classes have been defined to represent the basic arithmetic operations:

```
struct AddOp
{
    static inline double apply(const double & left, const double & y)
    {
        return x + y;
    }
};

struct MulOp
{
```

```
    static inline double apply(const double & left, const double & y)
    {
        return x * y;
    }
};

// ...and so on...
```

Note the use of the keyword `struct`; this simply signals that everything else in these classes---in particular, their default constructors and their destructors---are `public`.

Now suppose that a templated class `BinaryOp` has been defined as follows:

```
template<class Operator, class RHS>
class BinaryOp
{
  public :
    // empty constructor will be optimized away, but triggers
    // type identification needed for template expansion
    BinaryOp(
        Operator op,
        const Vector & leftArg,
        const RHS    & rightArg
    ) : left_(leftArg),
        right_(rightArg)
    {}

    // empty destructor will be optimized away
    ~BinaryOp()
    {}

    // calculate value of expression at specified index by recursing
    inline double apply(int i)
    {
        return Operator::apply(leftArg.apply(i), rightArg.apply(i));
    }

  protected :
    const Vector & left_;
    const RHS    & right_;
};
```

If `b` and `c` have been defined as `Vector`, and if `Vector::apply()` returns the vector element at the specified index, then when the compiler sees the following expression:

```
BinaryOp<MulOp, Vector, Vector>(MulOp(), b, c).apply(3)
```

it translates the expression into `b.apply(3) * c.apply(3)`. The creation of the intermediate instance of `BinaryOp` is optimized away completely, since all that object does is record a couple of references to arguments.

Why to go all this trouble? The answer is rather long, and requires a few seemingly-pointless steps. Consider what happens when the complicated expression above is nested inside an even more complicated expression, which adds an element of another vector `a` to the original expression's result:

```
BinaryOp< AddOp,
          Vector,
          BinaryOp< MulOp, Vector, Vector >
        >(a, BinaryOp< MulOp, Vector, Vector >(b, c)).apply(3);
```

This expression calculates `a.apply(3) + (b.apply(3) *c.apply(3))`. If the expression was wrapped in a `for` loop, and the loop's index was used in place of the constant `3`, the expression would calculate an entire vector's worth of new values:

```
BinaryOp< AddOp,
          Vector,
          BinaryOp< MulOp, Vector, Vector > >
        expr(a, BinaryOp< MulOp, Vector, Vector >(b, c));
for (int i=0; i<vectorLength; ++i)
{
  double tmp = expr.apply(i);
}
```

The possible nesting of `BinaryOp` inside itself is the reason that the `BinaryOp` template has two type parameters. The first argument to a `BinaryOp` is always a `Vector`, but the second may be either a `Vector` or an expression involving `Vector`s.

The code above is not something any reasonable person would want to write. However, having a compiler create this loop and its contained expression automatically is entirely plausible. The first step is to overload addition and multiplication for vectors, so that `operator+(Vector,Vector)` (and `operator*(Vector,Vector)`) instantiates `BinaryOp` with `AddOp` (and `MulOp`) as its first type argument, and invokes the `apply()` method of the instantiated object. The second step is to overload the assignment operator `operator=(Vector,Vector)` so that it generates the loop shown above:

```
template<class Op, T>
Vector & operator=(
    Vector & target,
    BinaryOp<Op> & expr
){
    for (int i=0; i<vectorLength; ++i)
    {
        target.set(i, expr.apply(i));
    }
    return target;
}
```

With these operator definitions in play, the simple expression:

```
Vector x, a, b, c;
// ...initialization...
x = a + b * c;
```

is automatically translated into the efficient loop shown above, rather than into the inefficient loops shown earlier. The expression on the right hand side is turned into an instance of a templated class whose type encodes the operations to be performed, while the implementation of the assignment operator causes that expression to be evaluated exactly once for each legal index. No temporaries are created, and only a single loop is executed.

This may seem complicated, but that's because it is. POOMA, and other libraries based on expression templates, push C++ to its limits because that's what it takes to get high performance. Defining the templated classes such a library requires is a painstaking task, as is ensuring that their expansion produces the correct result, but once it has been done, programmers can take full advantage of operator overloading to create compact, readable, maintainable programs without sacrificing performance.

[Prev] [Home] [Next]

POOMA Tutorial 1
A Laplace Solver Using Simple Jacobi Iteration

Contents:

Introduction
Laplace's Equation
A Sequential Solution
Using Intervals
Some Refinements
A Note on Affinity
Summary

Introduction

This tutorial introduces two of the most commonly used classes in POOMA: `Array`, which is used to store data, and `Interval`, which is used to specify a subsection of an array. The key ideas introduced in this tutorial are:

- the use of whole-array operations, such as scalar-to-array assignment and elementwise addition; and
- the use of intervals to specify array sections.

Laplace's Equation

Our first POOMA program solves Laplace's equation on a regular grid using simple Jacobi iteration. Laplace's equation in two dimensions is:

$$d^2V/dx^2 + d^2V/dy^2 = 0$$

where V is, for example, the electric potential in a flat metal sheet. If we approximate the second derivatives in X and Y using a difference equation, we obtain:

$$V(i, j) = (V(i+1, j) + V(i, j+1) + V(i-1, j) + V(i, j-1)) / 4$$

i.e. the voltage at any point is the average of the voltages at neighboring points. This formulation also gives us a way to solve this equation numerically: given any initial guess for the voltage V_0, we can calculate a new guess V_1 by using V_0 on the right hand side of the equation above. We can then use the calculated V_1 to calculate a new guess V_2, and so on.

This process, called Jacobi iteration, is the simplest in a family of relaxation methods than can be used to solve a wide range of problems. All relaxation methods iterate toward convergence, and use some kind of nearest-neighbor updating scheme, or *stencil*. The stencil for Jacobi iteration, for example, consists of five points arranged in a cross; other, larger stencils lead to different update rules, and different convergence rates. One of the main goals of POOMA was to make it easy for programmers to specify and implement stencil-based algorithms of this kind.

If we add charged particles to the system, we obtain Poisson's equation:

$$d^2V/dx^2 + d^2V/dy^2 = ß$$

where ß specifies the charge distribution. The solution to this equation can also be calculated using a relaxation method such as Jacobi iteration; the update equation is:

$$V(i, j)_{new} = (V(i+1, j) + V(i, j+1) + V(i-1, j) + V(i, j-1) - ß(i, j))/4$$

A Sequential Solution

Our first version of Jacobi iteration models a flat plate with a unit charge in its center using a 20×20 array. It uses POOMA's arrays conventionally, by looping over their elements, and is included in the release as `examples/Solvers/Sequential`. There is nothing wrong with using the library this way---POOMA's arrays are still very fast, and memory-efficient---but when compared with the refined program shown later, this code is longer and harder to read.

```
#include "Pooma/Arrays.h"
#include <iostream>

// The size of each side of the domain.
const int N = 20;

int
main(
    int                 argc,           // argument count
    char*               argv[]          // argument list
){
    // Initialize POOMA.
    Pooma::initialize(argc, argv);

    // The array we'll be solving for
    Array<2> V(N, N);

    // The right hand side of the equation (spike in the center)
    Array<2> b(N, N);

    // Initialize.
    for (int i=0; i<N; ++i){
        for (int j=0; j<N; ++j){
            V(i, j) = 0.0;
            b(i, j) = 0.0;
        }
    }
    b(N/2, N/2) = -1.0;

    // Iterate 200 times.
    Array<2> temp(N, N);
    for (int iteration=0; iteration<200; ++iteration)
    {
        // Use interior of V to fill temp
        for (int i=1; i<N-1; ++i){
            for (int j=1; j<N-1; ++j){
                temp(i, j) = 0.25*(V(i+1,j) + V(i-1,j) + V(i,j+1) + V(i,j-1) -
b(i,j));
            }
        }
        // Use temp to fill V
        for (int i=1; i<N-1; ++i){
            for (int j=1; j<N-1; ++j){
                V(i, j) = temp(i, j);
            }
        }
```

```
    }

    // Print out the result
    for (int j=0; j<N; ++j){
        for (int i=0; i<N; ++i){
            std::cout << V(i, j) << " ";
        }
        std::cout << std::endl;
    }

    // Clean up POOMA and report successful execution.
    Pooma::finalize();
    return 0;
}
```

Using Intervals

The program shown above is not much of an advance over its C equivalent. The programmer is still required to loop over data elements explicitly, even though these loops all take the same form. A better implementation of Jacobi iteration is shown below (and included in the release as `examples/Solvers/SimpleJacobi`). This version uses `Interval` objects to specify index ranges, which eliminates the need for the explicit loops of the first version.

```
01  #include "Pooma/Arrays.h"
02
03  #include <iostream>
04
05  // The size of each side of the domain.
06  const int N = 20;
07
08  int
09  main(
10      int                 argc,           // argument count
11      char*               argv[]          // argument list
12  ){
13      // Initialize POOMA.
14      Pooma::initialize(argc, argv);
15
16      // The array we'll be solving for
17      Array<2> V(N, N);
18      V = 0.0;
19
20      // The right hand side of the equation (spike in the center)
21      Array<2> b(N, N);
22      b = 0.0;
23      b(N/2, N/2) = -1.0;
24
25      // Specify the interior of the domain
26      Interval<1> I(1, N-2), J(1, N-2);
27
28      // Iterate 200 times
29      for (int iteration=0; iteration<200; ++iteration)
30      {
31          V(I,J) = 0.25*(V(I+1,J) + V(I-1,J) + V(I,J+1) + V(I,J-1) - b(I,J));
32      }
33
34      // Print out the result
```

```
35      std::cout << V << std::endl;
36
37      // Clean up POOMA and report success.
38      Pooma::finalize();
39      return 0;
40  }
```

The first three lines of this program include the header file needed to write POOMA programs, and the standard C++ I/O streams header file. `Pooma/Arrays.h` includes the header files that define POOMA's arrays. These arrays do not themselves contain data, but instead are handles on data containers through which programs can read, write, and apply operations to N-dimensional sets of elements. As we shall see, it is possible for many arrays to refer to the same underlying data in different ways.

`Pooma/Arrays.h` also includes all the declarations of the basic POOMA library interface functions. These general routines are used to initialize, query, and shut down the POOMA library environment, including the underlying run-time system.

The next statement in this program, at line 6, defines the size of our problem domain. In order to keep this code simple, this size is made a constant, and the array is made square. Real applications will usually employ variable-sized domains, and put off decisions about the actual sizes of arrays until run-time.

The function `Pooma::initialize()`, which is called at line 14, initializes some of POOMA's internal data structures. This function looks for certain POOMA-specific arguments in the program's command-line argument list, strips them out, and returns a possibly-shortened list. Programs should call `Pooma::initialize()` before calling any functions or methods from the POOMA library that might do operations in parallel. (They can alternatively use `Pooma::Options`, as described in another tutorial.) In practice, this means that it is generally a bad idea to declare POOMA objects as global variables, even if the program is not parallel when it is first written, since their presence can impede future portability.

Line 17 actually declares an array. The first thing to notice is that the rank of the array (i.e., the number of dimensions it has) is a template parameter to the class `Array`, while the initial dimensions of the array are given as constructor parameters. If we wanted to create a 3-dimensional array, we could change this line to be something like:

```
Array<3> V(SizeX, SizeY, SizeZ);
```

When the array `V` is created, POOMA realizes that some storage has to be created for it as well, and so it creates an actual data area at this point. When the assignment statement on the next line (line 18) is executed, POOMA sees an array target, but a scalar value, so it fills the whole array with the scalar's value.

Lines 21-23 create and initialize the array that stores the charge distribution term ß. This array's values are fixed: there is a single negative unit charge in the center of the domain, and no other charges anywhere else. Note how line 22 uses scalar-to-array assignment, while line 23 assigns to a single element of the array `b` using conventional subscripting. POOMA's arrays have many advanced features, but they also support mundane operations, such as reading or writing a particular location.

Line 26 introduces the `Interval` class. An `Interval` specifies a contiguous range of index values; the integer template argument to `Interval` specifies the interval's rank, while the constructor arguments specify the low and high ends of the interval's value. Thus, since `N` is fixed at 20 in this program, both `I` and `J` specify the one-dimensional interval from 1 to 19 inclusive.

Intervals are used to select sections of arrays using a Fortran 90-like syntax. Intervals and integers may be freely mixed when indexing an array; if any index in an expression is an interval, the result is a temporary alias for the specified array section. This alias is itself an array, since arrays are just lightweight handles on underlying data storage objects. The expression `V(I,J)` therefore returns a temporary array which aliases the interior of the same storage used by the array `V`.

Note that since the array `V` is square, the program could have declared a single `Interval` spanning 1..`N`-2, and used it to index `V` along both axes. However, the code is easier to read, and easier to modify to handle non-square domains, if two separate `Interval`s are used.

The loop spanning lines 29-32 performs the Jacobi relaxation. As discussed earlier, this consists of repeatedly averaging the charge distribution `b` at each location, and the values in `V` that are adjacent to that location, and then updating the location with that average. These calculations are done in parallel; that is, they appear to be calculated simultaneously for all

elements in the array. This is accomplished by using the `Interval`s declared on line 26 to select sections of `V` with appropriate offsets, and then relying on overloaded addition and subtraction operators to combine these sections. For example, the expression `V(I,J+1)` selects those elements `V(i,j)` of `V` for which `i` is in the range 1..`N`-2, and `j` is in the range 2..`N`-1 (i.e., the domain is offset in the second dimension by one). As can be inferred, arithmetic on `Interval`s works in the obvious way: for example, adding an integer adjusts all the elements of the `Interval` upward or downward.

Note that the assignment on line 31 automatically creates a temporary copy of the array `V`, so that values are not read while they are being overwritten. POOMA automatically detects cases in which the stencils on the reading side of an assignment overlap the stencils on the assignment's writing side, and creates temporaries as needed to avoid conflicts. The program shown in the next tutorial avoids the creation of temporaries simply by using non-overlapping stencils.

The statement on line 35 prints out the whole of the array `V`. POOMA overloads the usual stream operators (`<<` and `>>`) to handle most of the objects in the library sensibly. In this case, the output expression prints the elements of `V` a row at a time, putting each row on a separate line. Finally, line 38 calls the cleanup routine `Pooma::finalize()`, which complements the earlier call to `Pooma:initialize()` on line 14, and returns 0 to indicate successful completion.

Some Refinements

One thing that isn't shown in the program above is the precision of the calculations. To find out what this is, we can inspect the declaration of the class `Array` in the POOMA header file `Array.h`:

```
template < int Dim,
           class T         = POOMA_DEFAULT_ELEMENT_TYPE,
           class EngineTag = POOMA_DEFAULT_ENGINE_TYPE >
class Array : ...
{
    ...
};
```

The class `Array` has three template parameters: the number of dimensions, the element data type, and an engine tag that specifies how the underlying data is actually stored. We will discuss engines and engine tags in more detail in subsequent tutorials.

What makes this templated class declaration different from others we have seen so far is that default values are supplied for two of its three type parameters. The macros `POOMA_DEFAULT_ELEMENT_TYPE` and `POOMA_DEFAULT_ENGINE_TYPE` are defined in the header file `PoomaConfiguration.h`. The first specifies the default element type of arrays, while the second specifies their default storage mechanism. The default for the first is `double`, while the default for the second specifies dense, rectangular storage.

There are therefore two ways to change the precision of the calculations in the program above. One is to re-define `POOMA_DEFAULT_ARRAY_ELEMENT_T`:

```
#undef POOMA_DEFAULT_ELEMENT_TYPE
#define POOMA_DEFAULT_ELEMENT_TYPE float
#include "Pooma/Arrays.h"
```

The "undefinition" is needed because some compilers automatically read a "prefix file" before any other headers. This `#define` must come before any of POOMA's header files are included to ensure that all instantiations of all POOMA classes are done with the same default in effect.

The second, and more modular, way to change the precision of this Laplace solver is to specify the data types of the arrays explicitly:

```
Array<2, float> V(N, N);
Array<2, float> b(N, N);
```

This is generally considered better practice, as it is clear at the point of declaration what the data type of each array is, and because it makes it easier for programmers to combine classes that have been written independently. Other aspects of POOMA can and should be changed in the same way. (For example, the default engine type could be re-defined to make

parallel evaluation the default.)

It is also generally considered good practice to use `typedef`s to ensure the consistency of array definitions. For example, the Laplace solver could be written as follows:

```
typedef double LaplaceDataType_t;
typedef Array<2, LaplaceDataType_t> LaplaceArrayType_t;
LaplaceArrayType_t V(N, N);
LaplaceArrayType_t b(N, N);
```

Declaring types explicitly in this way might seem unnecessarily fussy in a small program such as this. However, all programs have a tendency to grow, and finding and modifying dozens of object declarations after the fact is much more tedious and error-prone than defining a type once, in one place, and then using it consistently through the rest of the program.

One final note on this program: it might seem cumbersome to declare the array on line 17, then initialize it with an assignment on the next line, instead of providing an initial value for the array's elements with an extra constructor argument. POOMA requires this in order to avoid ambiguity regarding what is a dimension, and what is an initial value. Since a single templated class `Array` is used for arrays of any dimension up to seven, it must provide constructors taking up to seven arguments which between them specify the array's dimensions. If we let `Sub1`, `Sub2`, and so on represent classes that can legally be used to specify dimensions (such as `int` or `Interval`), then `Array` must have constructors like the ones shown below:

```
template < int Dim,
           class T         = POOMA_DEFAULT_ELEMENT_TYPE,
           class EngineTag = POOMA_DEFAULT_ENGINE_TYPE >
class Array : ...
{
  public :
    template<class Sub1>
    Array(const Sub1& s1);

    template<class Sub1, class Sub2>
    Array(const Sub1& s1, const Sub2& s2);

    template<class Sub1, class Sub2, class Sub3>
    Array(const Sub1& s1, const Sub2& s2, const Sub3& s3);

    etc.
};
```

Suppose that `Array` also included constructors that took an initial value for the array's elements as an argument:

```
template < int Dim,
           class T         = POOMA_DEFAULT_ELEMENT_TYPE,
           class EngineTag = POOMA_DEFAULT_ENGINE_TYPE >
class Array : ...
{
  public :
    template<class Sub1>
    Array(const Sub1& s1, T initial_value);

    template<class Sub1, class Sub2>
    Array(const Sub1& s1, const Sub2& s2, T initial_value);

    etc.
};
```

Declarations such as the following would then be ambiguous:

```
Array<2, int> w(8, 5);
```

since the compiler would not be able to tell whether the two arguments were to be interpreted as dimensions, or as a dimension and an initializer. If C++ provided a way to "hide" constructors based on the value of a template argument, so that only the constructors for N-dimensional arrays could be called for `Array<N>`, this problem wouldn't arise. Since there is no such mechanism, POOMA requires programmers to specify initial values by wrapping them in a templated class. This is done as shown in the following declaration:

```
Array<2> w(5, 7, modelElement(3.14));
```

The function `modelElement()` does nothing except return an instance of `ModelElement<T>`, where `T` is the type of `modelElement()`'s argument. The `ModelElement` class in its turn only exists to provide enough type information for the compiler to distinguish between initializers and dimensions; the corresponding constructors of `Array` are:

```
template < int Dim,
           class T         = POOMA_DEFAULT_ARRAY_ELEMENT_T,
           class EngineTag = POOMA_DEFAULT_ARRAY_ENGINE >
class Array : ...
{
  public :

    // constructors for 1-dimensional arrays
    template<class Sub1>
    Array(const Sub1& s1);

    template<class Sub1>
    Array(const Sub1& s1, ModelElement<T> initial_value);

    // constructors for 2-dimensional arrays
    template<class Sub1, class Sub2>
    Array(const Sub1& s1, const Sub2& s2);

    template<class Sub1, class Sub2>
    Array(const Sub1& s1, const Sub2& s2, ModelElement<T> initial_value);

    etc.
};
```

Note that the function `modelElement()` is just a programming convenience: its only real purpose is to save programmers the trouble of typing:

```
Array<2> w(5, 7, ModelElement<double>(3.14));
```

A Note on Affinity

In some shared-memory machines, such as SGI Origins, every processor can access memory everywhere in the machine, but there is still a difference between "local" and "remote" memory. The memory chips are physically located with particular processors, so when processor 0 accesses memory that is actually stored with processor 127, the access is on average about 3-4 times slower than if processor 0 accesses its own memory. This only arises on very large machines---computers with up to 8 processors generally have truly symmetric memory.

When a program dynamically allocates memory on such a machine, the pages get mapped into the memory that is located with the CPU that first touches the memory. That is not necessarily the CPU that requested the allocation, since many pages could be allocated in one logical operation and pointers to them could be handed to other CPUs before being dereferenced.

Thus, both memory and threads can have an affinity for particular processors. A chunk of memory has affinity for a

particular CPU, and the thread scheduler can give a thread affinity for a CPU.

The difficulty that arises is that if the thread that is running the user's code initializes the memory for an `Array` with the `modelElement()` function mentioned in the first tutorial, all of the memory gets mapped to the CPU where that thread is running, instead of to a CPU across the machine.

One solution to this problem would be for the constructor that takes a `ModelElement` to generate the iterates that fill the memory, and then farm them out to the proper threads, so that the memory is mapped where the program actually wants it. This optimization is not in this release of POOMA, but will be considered for future releases.

Summary

This tutorial has shown that POOMA's `Array` class can be indexed sequentially, like a normal C or C++ array. It can also be indexed using Interval objects, each of which specifies a contiguous range of indices. When an `Array` is indexed using an `Interval`, the result itself acts like an array. Overloaded operators can be used to perform arithmetic and assignment on both arrays and selected array sections. Finally, the elementary data type of arrays can be changed globally by redefining a macro, or for individual arrays by overriding the default value of the `Array` template's second type parameter. The latter is considered better programming practice, particularly when `typedef` is used to localize the type definition.

[Prev] [Home] [Next]

POOMA Tutorial 2
Red/Black Update

Contents:

Introduction
Red-Black Update
Ranges
Engines
Passing Arrays to Functions
Calling the Function
A Note on Expressions
Using Two-Dimensional Ranges
Periodic Boundary Conditions
Operations and Their Results
Summary

Introduction

This tutorial shows how `Range` objects can be used to specify more general multi-valued array indices. It also introduces the `ConstArray` class, and delves a bit more deeply into the similarities between POOMA's arrays and the Standard Template Library's iterators.

Red-Black Update

Jacobi iteration is a good general-purpose relaxation method, but there are several ways to speed up its convergence rate. One of these, called red-black updating, can also reduce the amount of memory that a program requires. Imagine that the array's elements are alternately colored red and black, like the squares on a checkerboard. In even-numbered iterations, the red squares are updated using the values of their black neighbors; on odd-numbered iterations, the black squares are updated using the red squares' values. These updates can clearly be done in place, without any need for temporaries, and yield faster convergence for an equivalent number of calculations than simple Jacobi iteration.

A complete program that implements this is shown below (and is included in the release as `examples/Solvers/RBJacobi`). Its key elements are the declaration and initialization of two `Range` objects on line 37, the definition of the function that applies Jacobi relaxation on a specified domain on lines 9-17, and the four calls to that function on lines 43-47. The sections following the program source discuss each of these points in turn.

```
01  #include "Pooma/Arrays.h"
02
03  #include <iostream>
04
05  // The size of each side of the domain.
06  const int N = 20;
07
08  // Apply a Jacobi iteration on the given domain.
```

```
09  void
10  ApplyJacobi(
11      const Array<2>       & V,       // to be relaxed
12      const ConstArray<2>  & b,       // fixed term
13      const Range<1>       & I,       // range on first axis
14      const Range<1>       & J        // range on second axis
15  ){
16      V(I,J) = 0.25 * (V(I+1,J) + V(I-1,J) + V(I,J+1) + V(I,J-1) - b(I,J));
17  }
18
19  int
20  main(
21      int                 argc,           // argument count
22      char*               argv[]          // argument list
23  ){
24      // Initialize POOMA.
25      Pooma::initialize(argc, argv);
26
27      // The array we'll be solving for.
28      Array<2> V(N, N);
29      V = 0.0;
30
31      // The right hand side of the equation.
32      Array<2> b(N,N);
33      b = 0.0;
34      b(N/2, N/2) = -1.0;
35
36      // The interior domain, now with stride 2.
37      Range<1> I(1, N-3, 2), J(1, N-3, 2);
38
39      // Iterate 100 times.
40      for (int iteration=0; iteration<100; ++iteration)
41      {
42          // red
43          ApplyJacobi(V, b, I,   J);
44          ApplyJacobi(V, b, I+1, J+1);
45          // black
46          ApplyJacobi(V, b, I+1, J);
47          ApplyJacobi(V, b, I,   J+1);
48      }
49
50      // Print out the result.
51      std::cout << V << std::endl;
52
53      // Clean up and report success.
54      Pooma::finalize();
55      return 0;
56  }
```

Ranges

Our first requirement is a simple, efficient way to specify non-adjacent array elements. POOMA borrows the terminology of Fortran 90 and other data-parallel languages, referring to the spacing between a sequence of index values as the sequence's *stride*. For example, the sequence of indices {1,3,5,7} has a stride of 2, while the sequence {8,13,18,23,28} has a stride of 5, and the sequence {10,7,4,1} has a stride of -3.

POOMA programs represent index sequences with non-unit strides using `Range` objects. The templated class `Range` is a generalization of the `Interval` class seen in the previous tutorial (although for implementation reasons `Interval` is not derived from `Range`). When a `Range` is declared, the program must specify its rank (i.e., the number of dimensions it spans). The object's constructor parameters then specify the initial value of the sequence it represents, the upper bound on the sequence's value (or lower bound, if the stride is negative), and the actual stride. For example, the three sequences in the previous paragraph would be declared as:

```
Range<1> first ( 1,  7,  2);
Range<1> second( 8, 30,  5);
Range<1> third (10,  0, -3);
```

Note that the range's bound does not have to be a value in the sequence: an upward range stops at the greatest sequence element less than or equal to the bound, while a downward range stops at the smallest sequence element greater than or equal to the bound. This conforms to the meaning of the Fortran 90 triplet notation.

It may seem redundant to define a separate class for `Interval`, since it is just a `Range` with a stride of 1. However, the use of an `Interval` is a signal that certain optimizations are possible during compilation that take advantage of `Interval`'s unit stride. These optimizations cannot efficiently be deferred until the program is executing, since that would, in effect, require a conditional inside an inner loop. Another reason for making `Interval` and `Range` different classes is that `Interval`s can be used when declaring `Array` dimensions, but `Range`s cannot, since `Array`s must always have unit stride.

Engines

The previous tutorial said that the use of a non-scalar index as an array subscript selected a section from that array. The way this is implemented is tied into POOMA's notion of an engine. Arrays are just handles on engines, which are entities that give the appearance of managing actual data. Engines come in two types: *storage engines*, which actually contain data, and *proxy engines*, which can alias storage engines' data areas, calculate data values on demand, or do just about anything else in order to give the appearance there's an actual data area in there somewhere.

When an `Array` is declared, a storage engine is created to store that array's elements. When that array is subscripted with an `Interval` or a `Range`, the temporary `Array` that is created is bound to a view engine, which aliases the memory of the storage engine. Similarly, when an `Array` or `ConstArray` is passed by value to a function, the parameter is given a view engine, so that the values in the argument are aliased, rather than being copied. This happens in the calls to `ApplyJacobi()`, which is discussed below.

POOMA's engine-based architecture allows it to implement a number of useful tools efficiently. One of the simplest of these is the `ConstantFunction` engine, which provides a way to make a scalar behave like an array. For example, the following statements:

```
ConstArray<1, double, ConstantFunction> c(10);
c.engine().setConstant(3.14);
```

produce a full-featured read-only array that returns 3.14 for all elements. This is more efficient and uses less storage than making a `Brick` array with constant values. Engines that select components from arrays of structured types, or present arrays whose values are calculated on the fly as simple functions of their indices, are discussed in Tutorial 4 and Tutorial 6.

Passing Arrays to Functions

Lines 9-17 of this program define a function that applies Jacobi relaxation to a specified subset of the elements of an array. The actual calculation appears identical to that seen in the previous tutorial. However, the function's parameter declarations specify that `I` and `J` are `Range` objects, instead of `Interval`s. This means that the set of elements being read or written is guaranteed to be regularly spaced, although the actual spacing is not known until the program is run.

Another new feature in this function declaration is the use of the class `ConstArray`. Declaring something to be of type

`ConstArray` is not the same as declaring it to be a `const Array`. As mentioned earlier, POOMA's `Array` classes are handles on actual data storage areas. If something is declared to be a `const Array`, it cannot itself be modified, but the data it refers to can be. This is illustrated in line 16, which modifies the elements of `V` even though it is declared `const`. Put another way, the following is perfectly legal:

```
Array<1> original(10);
const Array<1>& reference = original;
reference(4) = 3.14159;
```

If an immutable array is really desired, the program must use the class `ConstArray`. This class overloads the element access method `operator()` to return a constant reference to an underlying data element, rather than a mutable reference. As a result, the following code would fail to compile:

```
Array<1> original(10);
ConstArray<1>& reference = original;
reference(4) = 3.14159;
```

since the assignment on its third line is attempting to overwrite a `const` reference. In fact, `Array` is derived from `ConstArray` by adding assignment and indexing that return mutable references. This allows an `Array` to be used as a `ConstArray`, but not vice versa. There is a subtle issue here though. One cannot initialize a `ConstArray` *object* with an `Array` *object*. The following code would fail to compile:

```
Array<1> a(10);
ConstArray<1> ca(a);
```

This problem results from a design decision to allow a `ConstArray` to be constructed with an arbitrary domain:

```
template<class Sub1>
ConstArray(const Sub1 & s1);
```

While an `Array` is a `ConstArray`, this function will be chosen by C++ compilers over the copy constructor because an exact match is preferred over a promotion to a base class. To avoid this problem, pass arrays by reference.

It is good programming practice to use `ConstArray` wherever possible, both because it documents the way the particular array is being used, and because it makes it harder (although not impossible) for functions to have inadvertent side effects.

It is important to note that the `Range` arguments to `ApplyJacobi()` must be defined as `const` references. The reason for this is that C++ does not allow programs to bind non-`const` references to temporary variables. For example, the following code is illegal:

```
void fxn(int& i)
{
    ....
}

void caller()
{
    int a = 5;
    fxn(a + 3);
}
```

Similarly, when the main body of the relaxation program adds offsets to the `Range` objects `I` and `J` on lines 44, 46, and 47, the overloaded addition operator creates a temporary object. `ApplyJacobi()` must therefore declare its corresponding arguments to be `const Range<1>&`.

The bottom line is that if a routine can get a temporary object, arguments should be passed by value or by `const` reference. If there is no possibility of the routine getting a temporary, arguments can be declared to be non-`const`

reference. For example:

```
template<int D, class T, class E>
void f(const Array<D, T, E>& a);

template<int D, class T, class E>
void g(Array<D, T, E> a);

template<int D, class T, class E>
void h(Array<D, T, E>& a);

void example()
{
    Interval<3> I(...);
    Array<3> x(...);

    f(x);                                   // OK
    g(x);                                   // OK
    h(x);                                   // OK
    f(x(I));                                // OK
    g(x(I));                                // OK
    h(x(I));                                // Bad, x(I) generates a temporary.
}
```

Note again that in the functions `f()`, `g()`, and `h()`, the array argument `a` can appear on the left hand side of an assignment. This is because `Array` is like an STL iterator: a `const` iterator or `const Array` can be dereferenced, it just can't be modified itself. If you want to ensure that the array itself can't be changed, use `ConstArray`.

Calling the Function

Lines 43-47 bring all of this together by passing the arrays `V` and `b` by value to `ApplyJacobi()`. The program makes four calls to this function; the first pair update the red array elements, while the second pair update the black array elements.

To see why two calls are needed to update each pair, consider the fact that each `Range` object specifies one half of the array's elements. The use of two orthogonal `Range`s therefore specifies $(1/2)^2=1/4$ of the array's elements. Simple counting rules of this kind are a useful check on the correctness of complicated subscript expressions.

As discussed above, each call to `ApplyJacobi()` constructs one temporary `Array` and one temporary `ConstArray`, each of which is bound to a view engine instead of a storage engine. Since these temporary objects are allocated automatically, they are also automatically destroyed when the function returns. POOMA uses reference counting to determine when the last handle on an actual area of array storage has been destroyed, and releases that area's memory at that time. Note that in this case, both arrays are bound to view engines, which do not have data storage areas of their own, so creating and destroying `ApplyJacobi()`'s arguments is very fast.

A Note on Expressions

As you may have guessed from the preceding discussion, POOMA expressions are first-class `ConstArray`s with an expression engine. As a consequence, expressions can be subscripted directly, as in:

```
Array<1> a(Interval<1>(-4, 0)), b(5), c(5);
for (int i = 0; i < 5; i++)
  c(i) = (a + 2.0 * b)(i);
```

This is equivalent, both semantically and in performance, to the loop:

```
for (int i = 0; i < 5; i++)
  c(i) = a(i - 4) + 2.0 * b(i);
```

Note that the offsetting of the non-zero-based arrays in expressions is handled automatically by POOMA.

POOMA also now includes a function called `iota()`, which allows applications to initialize array elements in parallel using expressions that depend on elements' indices. Instead of writing a sequential loop, such as:

```
for (i = 0; i < n1; ++i)
{
  for (j = 0; j < n2; ++j)
  {
    a(i,j) = sin(i)+j*5;
  }
}
```

a program could simply use:

```
a = sin(iota(n1,n2).comp(0)) + iota(n1,n2).comp(1)*5;
```

In general, `iota(domain)` returns a `ConstArray` whose elements are vectors, such that `iota(domain)(i,j)` is `Vector<2,int>(i,j)`. These values can be used in expressions, or stored in objects, as in:

```
Iota<2>::Index_t I(iota(n1,n2).comp(0));
Iota<2>::Index_t J(iota(n1,n2).comp(1));
a = sin(I*0.2) + J*5;
```

Using Two-Dimensional Ranges

As a general rule, whenever a set of objects are always used together, they should be combined into a single larger structure. If we examine the example program shown at the start of this tutorial, we can see that the two `Range` objects used to subscript arrays along their first and second axes are created in the same place, passed as parameters to the same function, and always used as a pair. We could therefore improve this program by combining these two objects in some way.

In POOMA, that way is to use a 2-dimensional `Interval` or `Range` instead of a pair of 1-dimensional `Interval`s or `Range`s. A 2-dimensional`Interval` is just the cross-product of its 1-dimensional constituents: it specifies a dense rectangular patch of an array. Similarly, a 2-dimensional `Range` is a generalization of the red or black squares on a checkerboard: the elements it specifies are regularly spaced, but need not have the same spacing along different axes.

An N-dimensional`Interval` is declared in the same way as its 1-dimensional cousin. An N-dimensional `Interval` is usually initialized by giving its constructor N 1-dimensional`Interval`s as arguments, as shown in the following example:

```
Interval<2> calc( Interval<1>(1, N), Interval<1>(1, N) );
```

Multi-dimensional POOMA arrays can be subscripted with any combination of 1-, 2-, and higher-dimensional indices, so long as the total dimensionality of those indices equals the dimension of the array. Thus, a 4-dimensional array can be subscripted using:

- four 1-dimensional indices
- a 2-dimensional index and a pair of 1-dimensional indices (in any order)
- a pair of 2-dimensional indices
- one 3-dimensional index and one 1-dimensional index (in any order); or
- a single 4-dimensional index.

If only a single array element is required, a new templated index class called `Loc` can be used as an index. Like other domain classes, this class can specify up to seven dimensions; unlike other domain classes, it only specifies a single location along each axis. Thus, the declaration:

```
Loc<2> origin(0, 0);
```

specifies the origin of a grid, while the declaration:

```
Loc<3> centerBottom(N/2, N/2, 0);
```

specifies the center of the bottom face of an $N \times N \times N$ rectangular block. `Loc` objects are typically used to specify key points in an array, or as offsets for specifying shifted domains. The latter of these uses is shown in the function `ApplyJacobi()` in the program below (which is included in the release as `examples/Solvers/RBJacobi`). This program re-implements the red/black relaxation scheme introduced at the start of this tutorial using 2-dimensional subscripting:

```
01  #include "Pooma/Arrays.h"
02
03  #include <iostream>
04
05  // The size of each side of the domain. Must be even.
06  const int N = 20;
07
08  // Apply a Jacobi iteration on the given domain.
09  void
10  ApplyJacobi(
11      const Array<2>       & V,       // to be relaxed
12      const ConstArray<2>  & b,       // fixed term
13      const Range<2>       & IJ       // region of calculation
14  ){
15    V(IJ) = 0.25 * (V(IJ+Loc<2>(1, 0)) + V(IJ+Loc<2>(-1,  0)) +
16                    V(IJ+Loc<2>(0, 1)) + V(IJ+Loc<2>( 0, -1)) - b(IJ));
17  }
18
19  int
20  main(
21      int                 argc,           // argument count
22      char*               argv[]          // argument vector
23  ){
24      // Initialize POOMA.
25      Pooma::initialize(argc, argv);
26
27      // The calculation domain.
28      Interval<2> calc( Interval<1>(1, N-2), Interval<1>(1, N-2) );
29
30      // The domain with guard elements on the boundary.
31      Interval<2> guarded( Interval<1>(0, N-1) , Interval<1>(0, N-1) );
32
33      // The array we'll be solving for.
34      Array<2> V(guarded);
35      x = 0.0;
36
37      // The right hand side of the equation.
38      Array<2> b(calc);
39      b = 0.0;
40      b(N/2, N/2) = -1.0;
```

```
41
42      // The interior domain, now with stride 2.
43      Range<2> IJ( Range<1>(1, N-3, 2), Range<1>(1, N-3, 2) );
44
45      // Iterate 100 times.
46      for (int i=0; i<100; ++i)
47      {
48          ApplyJacobi(V, b, IJ);
49          ApplyJacobi(V, b, IJ+Loc<2>(1, 1));
50          ApplyJacobi(V, b, IJ+Loc<2>(1, 0));
51          ApplyJacobi(V, b, IJ+Loc<2>(0, 1));
52      }
53
54      // Print out the result.
55      std::cout << V << std::endl;
56
57      // Clean up and report success
58      Pooma::finalize();
59      return 0;
60  }
```

The keys to this version of red/black relaxation are the `Interval` declarations on lines 28 and 31, and the array declarations on lines 34 and 38. The first `Interval` declaration defines the N-2 × N-2 region on which the calculation is actually done; the region defined by the second declaration pads the first with an extra column on each side, and an extra row on the top and the bottom. These extra elements are not part of the problem domain proper, but instead are used to ensure zero boundary conditions. Any other arbitrary boundary condition could be represented equally well by assigning values to these padding elements.

Using`Interval` objects that run from 1 to N-2 to specify the dimensions of the`Interval` object `calc` defined on line 28 means that when the array `b` is defined (line 38), its legal indices also run from 1 to N-2 along each axis. While POOMA uses 0..N-1 indexing by default, any array can have arbitrary lower and upper bounds along any axis, as this example shows. This is particularly useful when the natural representation for a problem uses a domain whose indices are in -N..N.

Note that line 31 could equally well have been written:

```
Interval<2> guarded(N, N);
```

In other words, integers work inside of `Domain` declarations the same way they do in `Array` declarations. If a program needs to declare a point, it can use:

```
Interval<2> x(Interval<1>(2, 2), Interval<1>(3, 2));
```

The declaration of `calc` on line 28 does need to be written as it is because the axes start at 1.

Examination of the update loop on lines 48-51, and the update assignment statement on lines 15-16, shows that the padding elements are never assigned to. Instead, the assignment on lines 15-16 only overwrites the interior of the array `V`. Note also that the domain used for the array `b`, which represents the fixed term in the Laplace equation, is only defined on the inner N-2 × N-2 domain. While the memory this saves is inconsequential in this 20×20 case, the savings grow quickly as the size and dimension of the problems being tackled increase.

Periodic Boundary Conditions

Our last look at red/black updating replaces the zero boundary condition of the previous examples with periodic boundaries in both directions. As is usual in programs of this kind, this is implemented by copying the values on one edge of the array into the padding elements next to the array's opposite edge after each relaxation iteration. For example, the padding elements to the right of the last column of the array are filled with the values from the first actual column of

the array, and so on. In the program shown below (included in the release as `examples/Solvers/PeriodicJacobi`), the "actual" values of the array `V` are stored in the region [1..N]×[1..N]. Elements with an index of either 0 or N+1 on either axis are padding, and are to be overwritten during each iteration.

The function that actually updates the periodic boundary conditions is called `ApplyPeriodic()`, and is shown on lines 20-28 below. The key to understanding this code is that when a "naked" integer is used to subscript a POOMA array, the result of that subscripting operation is reduced by one dimension in relation to that of the subscripted array. Thus, if a 2-dimensional array is subscripted using two specific integers, the result is a scalar value; if that same array is subscripted using an integer and a `Interval` or `Range`, the result is a 1-dimensional array.

Note that subscripting an `Array` with a `Loc<2>` yields a single scalar value, just as subscripting with two integers does, while subscripting with an `Interval` or `Range` that happens to refer to just one point yields an `Array` with just one element. There isn't a zero-dimensional `Array` (at least not in this release of POOMA), which is what the `Loc<2>` would have returned. The reduction in rank has to come from compile-time information, so `Loc` and integers reduce dimensionality, but`Interval` and `Range` do not.

```
01  #include "Pooma/Arrays.h"
02
03  #include <iostream>
04
05  // The size of each side of the domain. Must be even.
06  const int N = 18;
07
08  // Apply a Jacobi iteration on the given domain.
09  void
10  ApplyJacobi(
11      const Array<2>       & V,             // to be relaxed
12      const ConstArray<2>  & b,             // fixed term
13      const Range<2>       & IJ              // region of calculation
14  ){
15      V(IJ) = 0.25 * (V(IJ+Loc<2>(1,0)) + V(IJ+Loc<2>(-1,0)) +
16                      V(IJ+Loc<2>(0,1)) + V(IJ+Loc<2>(0,-1)) - b(IJ));
17  }
18
19  // Apply periodic boundary conditions by copying each slice in turn.
20  void
21  ApplyPeriodic(
22      const Array<2>       & V               // to be wrapped
23  ){
24      // Get the horizontal and vertical extents of the domain.
25      Interval<1> I = V.domain()[0],
26                  J = V.domain()[1];
27
28      // Copy each of the four slices in turn.
29      V(0,    J) = V(N, J);
30      V(N+1,  J) = V(1, J);
31      V(I,    0) = V(I, N);
32      V(I,  N+1) = V(I, 1);
33  }
34
35  int main(
36      int                 argc,           // argument count
37      char*               argv[]          // argument vector
38  ){
39      // Initialize POOMA.
40      Pooma::initialize(argc, argv);
41
```

```
42      // The calculation domain.
43      Interval<2> calc( Interval<1>(1, N), Interval<1>(1, N) );
44
45      // The domain with guard elements on the boundary.
46      Interval<2> guarded( Interval<1>(0, N+1), Interval<1>(0, N+1) );
47
48      // The array we'll be solving for.
49      Array<2> V(guarded);
50      V = 0.0;
51
52      // The right hand side of the equation.
53      Array<2> b(calc);
54      b = 0.0;
55      b(3*N/4,   N/4) = -1.0;
56      b(  N/4, 3*N/4) =  1.0;
57
58      // The interior domain, now with stride 2.
59      Range<2> IJ( Range<1>(1, N-1, 2), Range<1>(1, N-1, 2) );
60
61      // Iterate 200 times.
62      for (int i=0; i<200; ++i)
63      {
64          ApplyJacobi(V, b, IJ);
65          ApplyJacobi(V, b, IJ+Loc<2>(1,0));
66          ApplyJacobi(V, b, IJ+Loc<2>(0,1));
67          ApplyJacobi(V, b, IJ+Loc<2>(1,1));
68          ApplyPeriodic(V);
69      }
70
71      // Print out the result.
72      std::cout << V << std::endl;
73
74      // Clean up and report success.
75      Pooma::finalize();
76      return 0;
77  }
```

Note that, as we shall see in the next tutorial, the body of `ApplyPeriodic()` could more generally be written:

```
29  V(I.first(), J)         = V(I.last()-1,  J);
30  V(I.last(),  J)         = V(I.first()+1, J);
31  V(I,         J.first()) = V(I,           J.last()-1);
32  V(I,         J.last())  = V(I,           J.first()+1);
```

Operations and Their Results

One of the primary features of the POOMA array concept is the notion that "everything is an `Array`". For example, if you take a view of an `Array`, the result is a full-featured array. If you add two `Array`s together, the result is an `Array`. The table below illustrates this, using the declarations:

```
Array<2,Vector<2>> a
Array<2>           b
Interval<2>        I
Interval<1>        J
Range<2>           R
```

Operations Involving Arrays

Operation	Example	Output Type
Taking a view of the array's domain	`a()`	`Array<2,Vector<2>,BrickView<2,true>>`
Taking a view using an`Interval`	`a(I)`	`Array<2,Vector<2>,BrickView<2,true>>`
Taking a view using a `Range`	`a(R)`	`Array<2,Vector<2>,BrickView<2,false>>`
Taking a slice	`a(2,J)`	`Array<1,Vector<2>,BrickView<2,true>>`
Indexing	`a(2,3)`	`Vector<2>&`
Taking a read-only view of the array's domain	`a.read()`	`ConstArray<2,Vector<2>,BrickView<2,true>>`
Taking a read-only view using an`Interval`	`a.read(I)`	`ConstArray<2,Vector<2>,BrickView<2,true>>`
Taking a read-only view using a Range	`a.read(R)`	`ConstArray<2,Vector<2>,BrickView<2,false>>`
Taking a read-only slice	`a.read(2,J)`	`ConstArray<1,Vector<2>,BrickView<2,true>>`
Reading an element	`a.read(2,3)`	`Vector<2>`
Taking a component view	`a.comp(1)`	`Array<2,double,CompFwd<Engine<2,Vector<2>,Brick>,1>>`
Taking a read-only component view	`a.readComp(1)`	`ConstArray<2,double,CompFwd<Engine<2,Vector<2>,Brick>,1>>`
Applying a unary operator or function	`sin(a)`	`ConstArray<2,Vector<2>,ExpressionTag< UnaryNode<FnSin,ConstArray<2,Vector<2>,Brick>>>>`
Applying a binary operator or function	`a + b`	`ConstArray<2,Vector<2>,ExpressionTag< BinaryNode<OpAdd,ConstArray<2,Vector<2>,Brick>, ConstArray<2,double,Brick>>>>`

Indexing is the only operation that does not generate an `Array`. All other operations generate an `Array` or `ConstArray` with a different engine, perhaps a different element type, and, in the case of a slice, a different dimensionality. `ConstArray`s result when the operation is read-only.

Summary

This tutorial has shown that POOMA arrays can be subscripted using objects that represent index sequences with regular strides. Subscripting an array with a non-scalar index, or passing an array by value as a function parameter, creates a temporary array. While explicitly-declared arrays are bound to storage engines that encapsulate actual data storage, each temporary array is bound to a view engine, which aliases a storage engine's data area. Programs should use the templated class `ConstArray` to create immutable arrays, since the object created by a `const Array` declaration is actually an immutable handle on a mutable storage region. Finally, multi-dimensional and integer subscripts can be used to select subsections of arrays, and they yield results of differing dimensions.

[Prev] [Home] [Next]

POOMA Tutorial 3
Calculating Residuals

Contents:

Introduction
Implementing Reduction Using Loops
More on Domains
Some Subtleties
Using Built-In Reduction Functions
A Look Under the Hood
Summary

Introduction

It is easy to assign a scalar to an array in POOMA. Assigning an array to a scalar is a bit more complicated, since the array's values must be combined using some operator. Combinations of this kind are called *reductions*; common reduction operators include addition, maximum, and logical OR.

This tutorial shows how to perform reductions on arrays by querying their shape and size. Functions that do this are much more flexible than functions that rely on hard-coded dimensions and extents.

Implementing Reduction Using Loops

The programs shown so far have performed a fixed number of relaxation steps, with no regard for the actual rate at which the calculation is converging. A better strategy is to relax the system until the residual error is less than some threshold, while capping the number of iterations in order to avoid the program looping forever if it is given an ill-conditioned problem.

The program below (included in the release as `examples/Solvers/Residuals`) evaluates the residual error by summing the squares of the pointwise differences between the left and right sides of the usual update equation (line 16 in the code below). Lines 67-68 calculate this difference array, and pass it to the function `sum_sqr()`.

```
01  #include "Pooma/Arrays.h"
02
03  #include <iostream>
04
05  // The size of each side of the domain.
06  const int N = 20;
07
08  // Apply a Jacobi iteration on the given domain.
09  void
10  ApplyJacobi(
11      const Array<2>       & V,              // the domain
12      const ConstArray<2>  & b,              // constant condition
```

```
13      const Range<1>       & I,              // first axis subscript
14      const Range<1>       & J               // second axis subscript
15  ){
16      V(I,J) = 0.25 * (V(I+1,J) + V(I-1,J) + V(I,J+1) + V(I,J-1) - b(I,J));
17  }
18
19  // Calculate the sum of squares of all the elements in a 2D ConstArray.
20  template<class ValueType, class EngineTag>
21  ValueType sum_sqr(const ConstArray<2, ValueType, EngineTag> &A)
22  {
23      ValueType sum = 0;
24
25      int first_0 = A.domain()[0].first(),
26          last_0  = A.domain()[0].last(),
27          first_1 = A.domain()[1].first(),
28          last_1  = A.domain()[1].last();
29
30      for (int index_0=first_0; index_0<=last_0; ++index_0)
31      {
32          for (int index_1=first_1; index_1<=last_1; ++index_1)
33          {
34              ValueType value = A(index_0, index_1);
35              sum += value * value;
36          }
37      }
38      return sum;
39  }
40
41  int
42  main(
43      int                 argc,           // argument count
44      char*               argv[]          // argument list
45  ){
46      // Initialize POOMA.
47      Pooma::initialize(argc, argv);
48
49      // The array we'll be solving for.
50      Array<2> V(N, N);
51      V = 0.0;
52
53      // The right hand side of the equation.
54      Array<2> b(N, N);
55      b = 0.0;
56      b(N/2, N/2) = -1.0;
57
58      // The interior domain.
59      Interval<1> I(1, N-2), J(1, N-2);
60
61      // Iterate until converged, or a max of 1000 time steps.
62      double residual = 1.0; // anything greater than threshold
63      int iteration;
64      for (iteration=0; iteration<1000 && residual>1e-6; ++iteration)
65      {
66          ApplyJacobi(V, b, I, J);
67          residual = sum_sqr(V(I+1,J) + V(I-1,J) + V(I,J+1) + V(I,J-1)
68                             - (b(I,J) + 4.0*V(I,J)));
```

```
69      }
70
71      // Print out the result.
72      std::cout << "Iterations = " << iteration << std::endl;
73      std::cout << "Residual = "   << residual  << std::endl;
74      std::cout << V << std::endl;
75
76      // Clean up and report success.
77      Pooma::finalize();
78      return 0;
79  }
```

The function `sum_sqr()` takes a 2-dimensional array of arbitrary type, with an arbitrary engine, as its argument. Templating this function by the value type of the array means that the function can be used efficiently for arrays of other types, such as `int`, without any changes. Templating on the engine tag type is at least as important, for reasons that will be discussed below.

Line 23 declares the function's result variable. This declaration uses the type parameter `ValueType`, so that `sum_sqr` will work for arrays of any base type supporting addition, product, and assignment (the three operations applied to `sum` and the values read from the array). Note that `sum_sqr()` is defined to return a `ValueType` as well.

Lines 25-28 then determine the extent of the array `A`. The method `Array::domain()` returns an instance of the templated class `Domain`, which records the extent of the array's domain along each axis. Subscripting the result of `A.domain()` with 0 or 1 returns a temporary object that represents the size of the specified axis; the `first()` and `last()` calls on lines 25-28 therefore record the starting and ending indices of the array in both dimensions, regardless of the array's type or storage mechanism.

Lines 25-28 could also be written:

```
int first_0 = A.first(0),
    last_0  = A.last(0),
    first_1 = A.first(1),
    last_1  = A.last(1);
```

since `Array` provides short cuts for accessing domain extents.

It is important to note that the indices to `sum_sqr()`'s argument `A` are contiguous; that is, they run from 0 to an upper bound with unit stride in both dimensions. One of the many purposes of the `Array` class is to map logical, user-level indices to an actual data area. Once an array section has been selected by using `Interval`s or `Range`s as indices, that section appears to users to be compact and contiguous. Thus, the following (very contrived) code first sets every third element of a vector to 3, then sets every ninth element to 9, since the recursive call to `setThree()` selects every third element from its argument, which itself is every third argument of the original array:

```
const int N = 20;
Range<1> stride(0, N-1, 3);

void setThree(Array<1> a, double value, bool recurse)
{
    a = value;
    if (recurse){
        Range<1> newStride(0, (N-1)/3, 3);
        setThree(a(newStride), value*value, false);
    }
}

int main(int argc, char* argv[])
{
    Array<1> a(N);
```

```
    a = 0;
    setThree(a(stride), 3, true);
}
```

The rest of this function is straightforward. The nested loops beginning on line 30 traverse the array along both axes; the assignment on line 34 reads a value from the array, while that on line 35 accumulates the square of each value in `sum`.

More on Domains

POOMA provides a few useful shortcuts for working with domains, which can be used to generalize routines that manipulate arrays of varying sizes and shapes. The current version of the library provides three "wildcard" domains:

- `AllDomain<Dim>` does not take any constructor arguments. It is interpreted to mean "the whole of the relevant domain".
- `LeftDomain<Dim>`'s constructors take either a set of `Dim` integers, or a `Loc<Dim>`. It interprets these as the right endpoint of a new domain, and is used to specify a left (low-indexed) subdomain within a larger domain.
- `RightDomain<Dim>` is similar to `LeftDomain`, but is interpreted as the left endpoint of a (high-indexed) subdomain within a larger domain.

Wildcards are used to take a view of an existing `Array` in a way that is relative to the existing domain. For example, suppose a program has defined an `Array<2>` on the domain `[1:10:1, 5:8:1]`:

```
Array<2> A(Interval<1>(1,10), Interval<1>(5,8));
```

The following expression would take a view of this array that included the elements `[3:6:1, 5:8:1]` (i.e., only some of the first dimension, but all of the second):

```
A(Interval<1>(3,6), AllDomain<1>());
```

Note that the parentheses after `AllDomain<1>` are necessary because this statement is constructing an unnamed instance of this templated class.

If a program wants to take a view that starts with a given endpoint on one end, and uses the existing endpoint on the other end, it must use the `LeftDomain` or `RightDomain` wildcards. For example:

```
A(LeftDomain<1>(6), RightDomain<1>(7));
```

accesses the elements of `A` in the domain `[1:6:1, 7:8:1]`, where `A` is the same array declared above. Note that these wildcard domains are inclusive at both ends: `LeftDomain` uses the left portion of the existing domain, chopping it off at the given right endpoint, while `RightDomain` uses the right portion of the existing domain, starting from the given left endpoint.

Domain wildcards can be used in combination with `Array` methods such as `first()` and `last()` to get views that refer to just the left or right edges of a domain. For example:

```
A(LeftDomain<1>(A.first(0) + 1), AllDomain<1>())
```

refers to the width-2 domain on left edge of the first dimension of the array `A`.

Finally, `Array` and `ConstArray` overload `operator()`, and provide a method `read()`, to return a view of the array's entire domain. The `Array` version of `operator()` returns a writable view; otherwise, the view is read-only. These methods are useful because they allow programmers to write zero-based algorithms for arrays, no matter their domain. For example, the following copies elements of `b` into `a`:

```
Array<1> a(Interval<1>(-4, 0)), b(5);
for (int i = 0; i < 5; i++)
  a()(i) = b()(i);
```

Some Subtleties

Returning to the implementation of reductions once again, the most interesting thing about the `sum_sqr()` function is not its implementation, but what gets passed to it. The function call on lines 67-68 binds `sum_sqr()`'s argument `A` to the result of the expression:

```
V(I+1,J) + V(I-1,J) + V(I,J+1) + V(I,J-1) - (b(I,J) + 4.0*V(I,J)));
```

Most languages that support whole-array operations, such as Fortran 90, would create a full-sized temporary array by evaluating this expression at every point, and then pass that temporary variable to `sum_sqr()`. POOMA does not do this; what it does instead is the key to its high performance.

Recall once again that arrays in POOMA do not actually store data, but instead act as handles on engines that know how to return values given sets of indices. Some engines reference data storage directly; that is, they translate a set of indices into a value by looking up the value corresponding to those indices in memory. However, POOMA also contains *expression engines*, which use expression templates to calculate array values on demand.

When an expression like the one above is encountered, POOMA does *not* calculate all of its values at once. Instead, the expansion of the overloaded operators used in the expression creates an expression engine as the program is compiled. Whenever the array wrapper around this engine is subscripted, the engine calculates and returns the corresponding value.

This technique is called "lazy evaluation" and is the reason why the body of the inner loop of `sum_sqr()` is written as:

```
ValueType value = A(index_0, index_1);
sum += value * value;
```

If the body of this loop was instead written as:

```
sum += A(index_0, index_1) * A(index_0, index_1);
```

then the expression engine would evaluate the value of `A` at each location twice, since subscripting `A` is what triggers element evaluation.

The existence of expression engines is one of the reasons why `sum_sqr()`, and other functions that use POOMA, should template the engine type as well as the data type of their arguments. If the engine type of `sum_sqr()`'s `A` argument was not templated, it would default to `BrickEngine`, which is the engine that manages a dense, contiguous block of memory. The call on lines 67-68 would therefore be evaluated by constructing an expression engine (good), then evaluating it at each location in order to fill in the argument `A` (bad).

One mistake that is commonly made by programmers who are first starting to use POOMA is to forget that different arguments to a function can have different data or engine types. Consider, for example, a function whose job it is to compute the sum of the squares of the elementwise difference between two vectors. The natural way to write it (assuming that the lengths of the vectors are known to be the same) is:

```
template<class ValueType, class EngineTag>
ValueType sum_sqr_diff(
    const ConstArray<1, ValueType, EngineTag>& Left,
    const ConstArray<1, ValueType, EngineTag>& Right
){
    ValueType sum = 0;

    int first = Left.first(0),
        last  = Left.last(0);

    for (int index=first; index<=last; ++index)
    {
        ValueType value = Left(index) - Right(index);
```

```
            sum += value * value;
        }

        return sum;
    }
```

However, if `sum_sqr_diff()` is written this way, the compiler can only instantiate it when the data types *and* the engines of its arguments are exactly the same. This means that a call like:

```
Array<1, int>   intvec(10);
Array<1, float> floatvec(10);
double result = sum_sqr_diff(intvec, floatvec);
```

would fail to compile, since the template type argument `ValueType` cannot simultaneously match `int`, `float`, and `double`. Similarly, if one argument to `sum_sqr_diff()` was a plain old `Array`, while another argument was an expression, the compiler would either have to force full evaluation of the expression (in order to get something with the same engine type as the plain old array), or give up and report an error.

The most general way to define this function is as shown below. Both the data and engine types of the arguments are independent, so that the compiler has the degrees of freedom it needs to instantiate this function for a wide variety of arguments:

```
template<class LeftValueType,  class LeftEngineTag,
         class RightValueType, class RightEngineTag>
double sum_sqr_diff(
    const ConstArray<1, LeftValueType,  LeftEngineTag>&  Left,
    const ConstArray<1, RightValueType, RightEngineTag>& Right
){
    double sum = 0;

    int first = Left.first(0),
        last  = Left.last(0);

    for (int index=first; index<=last; ++index)
    {
        double value = Left(index) - Right(index);
        sum += value * value;
    }

    return sum;
}
```

But what's this? Everything important in this function is templated, except its `double` return type. If a user-defined extra-precision numerical type is used in the array arguments, the accumulator will have lower precision than the values being accumulated. Why isn't this function defined as:

```
// ILLEGAL
template<class LeftValueType,  class LeftEngineTag,
         class RightValueType, class RightEngineTag,
         class ReturnType>
ReturnType sum_sqr_diff(
    const ConstArray<1, LeftValueType,  LeftEngineTag>&  Left,
    const ConstArray<1, RightValueType, RightEngineTag>& Right
){
    ReturnType sum = 0;
    // body of function
    return sum;
```

```
}
```

so that the compiler can, for example, instantiate the function with a return type of `QuadPrecision` when presented with the following:

```
Array<2, QuadPrecision> A(N, N);
Array<2, double>        B(N, N);
initialization
QuadPrecision result = sum_sqr_diff(A, B);
```

The unfortunate answer is that overloading and templates in C++ doesn't work that way. To take a simple example, suppose an application has a set of functions for writing to a file, such as `put(char)`, `put(char*)`, and `put(int)`. When the code `put(x)` is seen, the compiler uses *only* the type of the argument `x` to figure out which function to call. There is no way for the compiler to distinguish between:

```
ostream s = put(x);
```

and

```
FILE* s=put(x);
```

because the return type of `put()` is *not* considered by the compiler during template instantiation. While there are good technical reasons for this, it is one of the biggest obstacles that the implementers of the POOMA library (and other templated libraries) have had to face. As the workarounds used in the library itself are too complex for these introductory tutorials, the best solution for newcomers to the library is either to use a high-precision type like `double`, or to use the type of one of the arguments to the function or method, and hope that it will be sufficiently precise.

Using Built-In Reduction Functions

The `sum_sqr()` function on lines 20-39 above uses object-oriented techniques to achieve generality, but still has the loops of a C or Fortran 77 program. These loops not only clutter the code, they also do not exploit any parallelism that the hardware this program is running on might offer. A much better solution is to use one of POOMA's built-in reduction functions, in this case `sum()`. The program that does this is included in the release as `examples/Solvers/Residual2`; the key change, to `sum_sqr()`, is shown below:

```
template<class ValueType, class EngineTag>
ValueType sum_sqr(const ConstArray<2, ValueType, EngineTag>& A)
{
    return sum(A * A);
}
```

As might be expected, POOMA provides many other reduction functions:

`sum`	sum all the elements in an array
`prod`	multiply all of the elements in an array
`max`	find the maximum value in an array
`min`	find the minimum value in an array
`all`	returns true if all of the array's elements are non-zero
`any`	returns true if any of the array's elements are non-zero
`bitOr`	does a bitwise or of all of the elements
`bitAnd`	does a bitwise and of all of the elements

Note that since names such as `bitor` and `bitand` are actually reserved keywords in C++, some of these functions have names such as `bitOr` and `bitAnd`.

POOMA presently puts its reduction operators, along with most of the other things it defines, in the global namespace.

Since all of these operators are templated on POOMA classes, there is very little chance of collision with other functions with the same names. While it would be better programming practice to put everything into a namespace, like POOMA's `initialize()` and `finalize()` functions, some compilers still have trouble with the combination of templates and namespaces. Once these compilers are brought up to full ANSI/ISO compliance, all POOMA functions and classes will be placed in the `Pooma::` namespace.

Of course, the obvious next step is to get rid of `sum_sqr()` entirely, and move the residual calculation into the main loop:

```
Array<2> temp;
for (iteration=0; iteration<1000 && residual>1e-6; ++iteration)
{
    ApplyJacobi(V, b, I, J);
    temp = V(I+1,J) + V(I-1,J) + V(I,J+1) + V(I,J-1) - (b(I,J) + 4.0*V(I,J));
    residual = sum(temp * temp);
}
```

This is tempting, but wrong: if you compare the performance of this version of the program to that of the original, you will find that this one is significantly slower. The reason is that the assignment to `temp` in the middle of the loop above does not create an expression engine, but instead allocates and fills in an array. Only by passing an expression to a templated function can a program give the compiler an opportunity to capture enough information about the expression to create an array which is bound to an expression engine.

Life would clearly be better if there was some way to declare temporary array variables that were guaranteed to be bound to expression engines, instead of storage engines. However, in order to do this, programmers would have to specify the type of the temporary array exactly. By the time the residual expression above has been expanded, its type definition is several thousand characters long; the complexity of the types of longer expressions grows very, very quickly.

A Look Under the Hood

By now, you may be curious about how POOMA does what it does. This section therefore takes a look at the implementation of reduction operators; while many details are omitted, it should give you some idea of how the library is structured, and why some of its features appear the way they do.

We have three requirements for a global reduction function such as `sum()`: it must be able to reduce arrays of arbitrary size, it must be able to reduce arrays of arbitrary type, and it must efficiently use the same underlying machinery as other reductions. The first two criteria need no justification; the third one is a software engineering concern. If each reduction function has to be completely self-contained (i.e., if all of the parallelization and looping code has to be duplicated), then maintaining the library will be difficult. On the other hand, we cannot afford to write a generic reduction routine that takes a function pointer or an object with a virtual method as an argument, since the cost of indirection inside an inner loop is unacceptable.

POOMA's authors solve these problems by using a *trait class* to represent each primitive reduction operation. A trait class is a class whose only purpose is to be used to instantiate other templated classes. Each class in a family of trait classes defines constants, enumeration elements, and methods with identical names and signatures, so that they can be used interchangeably.

Trait classes are not part of the C++ language definition, but are instead a way of using template instantiation as an abstraction mechanism in yet another way. Instead of overriding virtual functions inherited from parent classes, templated classes can use constants and methods supplied by template parameters in generic ways. For example, either of the classes `Red` and `Green` in the code below can be used to instantiate `Blue`, but when the values of those different instances are printed, they display different values:

```
struct Red
{
    enum { Val = 123; }
};
```

```
struct Green
{
    enum { Val = 456; }
};

template<class T>
class Blue
{
  public:
    Blue() : val_(T::Val) {}
    const unsigned int val_;
};

int main()
{
    Blue<Red>   br;
    Blue<Green> bg;
    std::cout << br.val_ << std::endl;
    std::cout << bg.val_ << std::endl;
    return 0;
}
```

Note that the example above declares `Red` and `Green` as `struct`s instead of as `class`es. The only difference between the two kinds of declarations is that a `struct`'s members are `public` by default, while those of a `class` are `private`. This saves one line each in the definitions of `Red` and `Green`.

POOMA defines a family of trait classes representing the C++ assignment operators. These classes, which are part of the Portable Expression Template Engine (PETE), are defined as follows:

```
PETE_DEFINE_ASSIGN_OP((a = b),   OpAssign)
PETE_DEFINE_ASSIGN_OP((a += b),  OpAddAssign)
PETE_DEFINE_ASSIGN_OP((a -= b),  OpSubtractAssign)
PETE_DEFINE_ASSIGN_OP((a *= b),  OpMultiplyAssign)
PETE_DEFINE_ASSIGN_OP((a /= b),  OpDivideAssign)
PETE_DEFINE_ASSIGN_OP((a %= b),  OpModAssign)
PETE_DEFINE_ASSIGN_OP((a |= b),  OpBitwiseOrAssign)
PETE_DEFINE_ASSIGN_OP((a &= b),  OpBitwiseAndAssign)
PETE_DEFINE_ASSIGN_OP((a ^= b),  OpBitwiseXorAssign)
PETE_DEFINE_ASSIGN_OP((a <<= b), OpLeftShiftAssign)
PETE_DEFINE_ASSIGN_OP((a >>= b), OpRightShiftAssign)
```

where each use of the macro `PETE_DEFINE_ASSIGN_OP` defines a `struct` with the same members:

```
#define PETE_DEFINE_ASSIGN_OP(Expr,Op)                      \
struct Op {                                                 \
    Op() {}                                                 \
                                                            \
    Op(const Op&) {}                                        \
                                                            \
    enum {                                                  \
        tag = BinaryUseLeftTag                              \
    };                                                      \
                                                            \
    template<class T1, class T2>                            \
    inline typename BinaryReturn<T1, T2, Op>::Type_t        \
    operator()(                                             \
```

```
        T1&       a,                                                \
        const T2& b                                                 \
    ) const {                                                       \
        return Expr;                                                \
    }                                                               \
};
```

This `struct` has four members: a default constructor, a copy constructor, a constant (defined using an `enum`) called `tag`, and a templated method `operator()`. (The default and copy constructors might appear to be unnecessary, but omitting them results in Uninitialized Memory Read (UMR) warnings from memory checking tools such as Purify.) As discussed earlier, templated methods are instantiated when and as required, just like templated functions, but are still members of their containing class. In this case, the templated `operator()` takes a destination argument `a` of one type, and a source argument `b` of another type, and performs the expression `Expr` (such as "`a+=b`") on them.

Thus, whenever `OpAddAssign` or another class in this family of trait classes is used in an expression, the templated method `operator()` is instantiated with the appropriate types (such as `int` for the source, and `double` for the destination). Since this method is not virtual, there is no abstraction penalty: code using any particular instantiation of this templated method will run at maximum speed.

The only feature of this macro that has not yet been explained is the use of `BinaryReturn`. This is another trait class, whose only purpose is to define the type of the result of applying an operation `Op` to values of types `T1` and `T2`. For example, `BinaryReturn<int,float,OpAdd>` defines `Type_t` to be `float`, while `BinaryReturn<double,float,OpMul>` defines `Type_t` to be `double`. Again, POOMA uses template instantiation as an abstraction mechanism, so that logically repetitive code does not have to be physically replicated. (This is an illustration of how to solve the problem discussed earlier of how to generalize the return type of a function so that it is adequate for the input argument types.)

With all of this machinery in place, `sum()` is now easy to build:

```
template<int Dim, class T, class EngineTag>
inline T sum(
    const ConstArray<Dim, T, EngineTag>& a
){
    return globalReduction(a, T(0), OpAddAssign());
}
```

The function just passes the array, a value to initialize the sum with, and the reduction operation to be applied to a generic templated function called `globalReduction()`. The first argument to this function is the array; the second is an initialization value (which is also the value returned if the array is empty), and the third specifies the operation to apply. By the time `globalReduction()` is expanded, `OpAddAssign::operator()` has been inlined by POOMA's expression template machinery. Note that the initialization value must be an identity element for the operation; while zero works in most cases, some operations (such as logical and bitwise OR) must use other values.

Summary

POOMA achieves high performance using expression engines, which are constructed automatically during compilation, and which evaluate complex array expressions on demand in order to avoid creation of temporary arrays. In order to support mixed data types, and the use of both arrays and array expressions as arguments, user-defined functions should be templated separately by both the data type and engine type of all of their arguments. Unfortunately, C++ does not support templatization on function return type, which can make it difficult to write fully-generic functions. Finally, POOMA provides several built-in reduction functions, such as summation, product, and logical combination. These are implemented using a generic framework, which can be extended by knowledgeable users.

[Prev] [Home] [Next]

POOMA Tutorial 4
Further Topics

Contents:

Introduction
Block And Evaluate
Small Vectors and Tensors
A Note on Tensor Accumulators
Using Multiple Patches
Guard Layers
Taking Layout Into Account
Component Views
Summary

Introduction

One of POOMA's most attractive features is its high performance on both single-processor and shared-memory multiprocessor machines. As future releases of the library will also support distributed-memory multicomputers and networks of workstations, POOMA's authors have had to think very carefully about how to obtain the best possible performance across a wide range of applications on different architectures.

The heart of the problem POOMA's authors face is that while data-parallel programming is a natural way to express many scientific and numerical algorithms, straightforward implementations of it do exactly the wrong thing on modern RISC architectures, whose performance depends critically on the re-use of data loaded into cache. If a program evaluates `A+B+C` for three arrays `A`, `B`, and `C` by adding `A` to `B`, then adding `C` to that calculation's result, performance suffers both because of the overhead of executing two loops instead of one, but also (and more importantly) because every value in the temporary array that stores the result of `A+B` has to be accessed twice: once to write it, and once to read it back in. As soon as this array is too large to fit into cache, the program's performance will drop dramatically.

The first section of this tutorial explains what POOMA does to solve this problem. Subsequent sections discuss other advanced aspects of POOMA, such as reduction functions that will execute efficiently regardless of how arrays are laid out in memory, and the use of traits classes to provide programs with information about POOMA objects.

Block And Evaluate

POOMA tries to resolve the tension between how programmers want to express their ideas, and what runs most efficiently on modern architectures, by delaying the evaluation of expressions until enough is known about their context to ensure that they can be evaluated efficiently. It does this by blocking calculations into units called iterates, and putting those iterates into a queue of work that is still to be done. Each iterate is a portion of a computation, over a portion of a domain. POOMA tracks data dependencies between iterates dynamically to ensure that out-of-order computation cannot occur.

Depending on the switches specified during configuration when the library is installed, and the version of the POOMA library that a program is linked against, POOMA will run in one of four different modes. In the first mode, the work queue doesn't actually exist. Instead, the single thread of execution in the program evaluates iterates as soon as they are "queued" (i.e., work is done immediately). The result is that all of the calculations in a statement are completed by the time execution reaches the semi-colon at the end of that statement.

In its second mode, POOMA maintains the work queue, but all work is done by a single thread. The queue is used because the explicit parceling up of work into iterates gives POOMA an opportunity to re-order or combine those iterates. While the overhead of maintaining and inspecting the work queue can slow down operations on small arrays, it makes operations on large arrays much faster.

For example, consider the four function calls that perform red/black relaxation in the second tutorial. In order to get the highest possible performance out of the cache, all four of these expressions should be evaluated on a given cache block before any of the expressions are evaluated for the next cache block. Managing this by hand is a nightmare, both because cache size varies from machine to machine (even when those machines come from the same manufacturer), and because very slight changes in the dimensions of arrays can tip them in or out of cache. POOMA's array classes and overloaded operators do part of the job by creating appropriately-sized iterates; its work queue does the rest of the job by deciding how best to evaluate them. The net result is higher performance for less programmer effort.

POOMA's third and fourth modes of operation are multi-threaded. Each thread in a pool takes iterates from the work queue when and as they become available. Iterates are evaluated independently; the difference between the two modes is that one is synchronous and blocks after evaluating each data-parallel statement, while the other is asynchronous and permits out-of-order execution. The table below summarizes these four modes, along with the configuration arguments used to produce each.

1. Synchronous Serial Conventional sequential execution `--serial`	**2. Asynchronous Serial** Serial work queue `--parallel --sched serialAsync`
3. Synchronous Parallel Multithreaded, blocking after each data-parallel statement `--parallel --sched sync`	**4. Asynchronous Parallel** Multithreaded, out-of-order execution `--parallel --sched async`

A very important function in POOMA's work allocation system is `Pooma::blockAndEvaluate()`. It is one of only two functions that expose the library's internal parallelism and cache optimizations to users. While POOMA automatically calls it at the right times in most cases, there are a few situations in which programmers should call it explicitly.

If evaluation has been deferred, the statements being evaluated are not guaranteed to have completed until `blockAndEvaluate()` is called. POOMA does this by itself inside of `operator<<`, reductions, and so on, but there is a place where the performance overhead of doing that check would be so high as to be unacceptable: indexing with integers. If `blockAndEvaluate()` was called inside every use of `operator()`, it would be impossible to write serial loops efficiently.

This means that when a program is running in modes 2-4 (i.e., using either parallelism or potentially asynchronous execution), it must call `blockAndEvaluate()` before subscripting arrays with integers. Failure to do so can lead to race conditions, and other hard-to-find errors.

Typical uses of `blockAndEvaluate()` look like:

```
Array<2> A(N, N);
A = 0;
Pooma::blockAndEvaluate();
A(N/2, N/2) = 1;
```

or:

```
Loc<2> center(N/2, N/2);
Pooma::blockAndEvaluate();
A(center) = 1;
```

Without the calls, the code might not parallelize correctly. If, however, code like the following is used instead:

```
Interval<2> center(Interval<1>(N/2, N/2), Interval<1>(N/2, N/2));
A(center) = 1;
```

then correct execution is guaranteed, because this assignment will be handled using all of POOMA's parallel machinery. Of course, the safe version is somewhat slower, and should not be used inside a time-critical loop, since it would implicitly be doing locking and unlocking on every call.

It can be very tedious to place `blockAndEvaluate()` calls in code that mixes scalar and data-parallel statements. It is easier and less error-prone to simply turn off asynchronous operation temporarily. This is accomplished by calling `Pooma::blockingExpressions(true)` at the beginning of such a block and then calling `Pooma::blockingExpressions(false)` at the end.

Small Vectors and Tensors

POOMA includes two "tiny" classes that are optimized to represent small vectors and tensors. Not surprisingly, these are called `Vector` and `Tensor`; their declarations are:

```
template<int Size, class T = double, class EngineTag = Full>
struct Vector;

template<int Size, class T = double, class EngineTag = Full>
struct Tensor;
```

The size parameters specify the fixed size(s) of the objects, and are used as follows:

```
Vector<3> v;         // 3-component vector of doubles.

Vector<2, int> vi;   // 2-component vector of ints.

Tensor<2, int> t;    // 2×2 tensor of ints.
```

Note that these classes use engine abstractions, just like their grown-up `Array` counterpart. The only engine class available for `Vector` in this release is `Full`, which signals that all elements of the vector or tensor are stored. For `Tensor`, in addition to `Full`, POOMA provides `Antisymmetric`, `Symmetric`, and `Diagonal` classes to use for the `EngineTag` parameter. The names of these classes describe their mathematical meaning. In the following table, we show the definitions of the tensor symmetries, indexing convention, and the way the data values are stored internally in the `Tensor<Dim,T,EngineTag>` classes. Note that we only store values that cannot be computed from other values, but the user can still index non-`Full` `Tensor` objects as if they had all elements stored.

EngineTag Value	Tensor Structure	(i,j) Indices	Array Storage of Elements

`Full`	`| x00 x01 x02 |`
	`| x10 x11 x12 |`
	`| x20 x21 x22 |`
		`| 0,0 0,1 0,2 |`
		`| 1,0 1,1 1,2 |`
		`| 2,0 2,1 2,2 |`
			`| x_m[0] x_m[3] x_m[6] |`
			`| x_m[1] x_m[4] x_m[7] |`
			`| x_m[2] x_m[5] x_m[8] |`

`Symmetric`	`| x00 x10 x20 |`
	`| x10 x11 x21 |`
	`| x20 x21 x22 |`
		`| 0,0 0,1 0,2 |`
		`| 1,0 1,1 1,2 |`
		`| 2,0 2,1 2,2 |`
			`| x_m[0] |`
			`| x_m[1] x_m[3] |`
			`| x_m[2] x_m[4] x_m[5] |`

`Antisymmetric`	`| 0 -x10 -x20 |`
	`| x10 0 -x21 |`
	`| x20 x21 0 |`
		`| 0,0 0,1 0,2 |`
		`| 1,0 1,1 1,2 |`
		`| 2,0 2,1 2,2 |`
			`|  |`
			`| x_m[0] |`
			`| x_m[1] x_m[2] |`

`Diagonal`	`| x00 0 0 |`
	`| 0 x11 0 |`
	`| 0 0 x22 |`
		`| 0,0 0,1 0,2 |`
		`| 1,0 1,1 1,2 |`
		`| 2,0 2,1 2,2 |`
			`| x_m[0] |`
			`| x_m[1] |`
			`| x_m[2] |`

The code below (included in the release as `examples/Tiny`) is a short example of how `Vector` and `Tensor` classes can be used:

```
01 #include "Pooma/Arrays.h"
02
```

```
03 int main(
04     int                 argc,           // argument count
05     char*               argv[]          // argument list
06 ){
07  // Initialize POOMA.
08  Pooma::initialize(argc, argv);
09
10  // Make an array of 100 3D ray vectors.
11  Loc<1> patchSize(25);
12  UniformGridLayout<1> layout(Interval<1>(100), patchSize);
13  Array< 1, Vector<3>, MultiPatch<UniformTag,Brick> > rays(layout);
14
15  // Set the third component of all of the vectors to zero.
16  rays.comp(2) = 0.0;
17
18  // Starting some scalar code, must block.
19  Pooma::blockAndEvaluate();
20
21  // Fill the vectors with a random value for the first component.
22  for (int i = 0; i<100; i++)
23  {
24    rays(i)(0) = rand() / static_cast<double>(RAND_MAX);
25  }
26
27  // Define a unit vector pointing in the y direction.
28  Vector<3> n(0.0, 1.0, 0.0);
29
30  // Set the second component so that the length is one.
31  rays.comp(1) = sqrt(1.0 - rays.comp(0) * rays.comp(0));
32
33  // Reflect the rays off of a plane perpendicular to the y axis.
34  rays += -2.0 * dot(rays, n) * n;
35
36  // Define a diagonal tensor:
37  Tensor<3,double,Diagonal> xyflip2(0.0);
38  xyflip2(0,0) = -2.0;
39  xyflip2(1,1) = -2.0;
40
41  // Tensor-Vector dot product multiplies x and y components by -2.0:
42  rays = dot(xyflip2, rays);
43
44  // Output the rays.
45  std::cout << rays << std::endl;
46
47  // Clean up and leave.
48  Pooma::finalize();
49  return 0;
50 }
```

As line 13 of this code shows, programs can declare POOMA `Array`s with elements other than basic arithmetic types like `int` or `double`. In particular, `Vector`, `Tensor`, and `complex` are explicitly supported. Please contact pooma@acl.lanl.gov for information on using other, more complicated types.

The `Array::comp()` method used on line 16 does *component forwarding*. The expression `rays.comp(2)` returns an `Array<double>` that supports writing into the second component of each vector element of rays. This is a data-parallel statement that works in a way analogous to the loop at lines 22-25, except that the POOMA evaluator will calculate patches in parallel. Thus, if a program had an array of tensors `T`, it could change the element in the 0th row, 1st column with `T.comp(0, 1)`. Note that, unlike `Array`, both `Vector` and `Tensor` always index from zero.

Line 24 shows that, as expected, the i[th] component of a `Vector V` can be accessed for both reading and writing using the syntax `V(i)`; `Tensor` element access requires two subscripts. Thus, the first subscript in the expression `rays(i)(0)` returns the i[th] element of the `Vector rays`, while the second subscript returns the zeroth component of that vector. Component forwarding is intimately related to the notion of component views, which are discussed below.

Line 28 shows that `Vector`s can be initialized with `Size` element values. Similarly, instances of `Tensor` can be initialized with `Size*Size` element values.

The data-parallel expression on line 31 shows that the usual math functions can be applied to entire arrays. The unary and binary functions supported are:

`acos`	`asin`	`atan`	`ceil`	`cos`	`cosh`
`exp`	`fabs`	`floor`	`log`	`log10`	`sin`
`sinh`	`sqrt`	`tan`	`tanh`	`imag`	`real`
`abs`	`arg`	`norm`[1]			
`ldexp`	`pow`	`fmod`	`atan2`	`dot`[2]	`polar`[3]

1. `complex<T>` only
2. `Vector` and `Tensor` only
3. `complex<T>` only

Line 34 is a data-parallel expression on vectors. In addition to dot product, the normal arithmetic functions involving `Vector` and `Tensor` are supported (see the note on tensor accumulators below for exceptions), as are the following named functions on vectors and tensors:

`norm(Vector<D,T,E> &v)`:

Returns a scalar of type `T`, equal to `sqrt(dot(v, v))`.

`norm2(Vector<D,T,E> &v)`:

Returns a scalar of type `T`, equal to `dot(v, v)`.

`trace(Tensor<D,T,E> &t)`:

Returns a scalar of type `T`, equal to the trace of the tensor `t` (sum of diagonal elements).

`det(Tensor<D,T,E> &t)`:

Returns a scalar of type `T`, equal to the determinant of the tensor `t`.

`transpose(Tensor<D,T,E> &t)`:

Returns a tensor of type `Tensor<D,T,E>`, equal to the transpose of the tensor (element (*i*,*j*) of the transpose is equal to element (*j*,*i*) of the input tensor `t`.

```
template<class OutputEngineTag, int D, class T, class EngineTag>
Tensor<D,T,OutputEngineTag> &symmetrize(Tensor<D,T,E> &t):
```

Returns a tensor of type `Tensor<D,T,E>`, applying a n appropriate symmetrizing operation to convert from the symmetry of the input `EngineTag` (for example, `Full`) to the symmetry of the `OutputEngineTag` (for example, `Antisymmetric`. This is invoked using explicit template instantiation for the desired `OutputEngineTag`. For example:

```
Tensor>2,double,Full> t(1.0, 2.0, 3.0, 4.0);
Tensor>2,double,Antisymmetric> at = symmetrize<Antisymmetric>(t);
std::cout << " t = " << t << std::endl;
std::cout << "at = " << at << std::endl;
```

produces the output:

```
t = ( ( 1 3 )( 2 4 ) )
a = ( ( 0 0.5 )( -0.5 0 ) )
```

`dot(Vector&, Tensor&)`:

Returns a `Vector` via matrix-vector product of the arguments.

`dot(Tensor&, Vector&)`:

Returns a `Vector` via matrix-vector product of the arguments.

`dot(Tensor&, Tensor&)`:

Returns a `Tensor` via matrix-matrix product of the two arguments.

`outerProduct(Vector&, Vector&)`:

Returns a `Tensor` (with `EngineTag=Full`) via outer (tensor) product of the arguments.

These functions also operate on `Array`s of `Tensor` and `Vector` elements (and `DynamicArray`s, and `Field`s.)

Lines 37-39 show construction of a diagonal tensor using the `Tensor` class with `Diagonal` for the `EngineTag` parameter; line 37 constructs it with all (diagonal) values equal to 0.0, then lines 38 and 39 assign the first two elements along the diagonal to -2.0. Line 42 illustrates the `Tensor`-`Vector` dot product, returning a `Vector`.

This release of POOMA does not offer double-dot products, cross products or any other vector or tensor operations; these are being considered for future releases.

Finally, as line 45 shows, arrays of vectors can be output like arrays of any other type.

A Note on Tensor Accumulators

Accumulation operators such as `operator*=()` acting on `Tensor<D,T,EngineTag>` may result in a `Tensor` having different symmetry (different `EngineTag` than what you are accumulating into. For example,

```
Tensor<2,double,Antisymmetric> t1, t2;
// ... assign values
t1 *= t2;
```

is incorrect, because the result of multiplying the two antisymmetric tensors would be a *symmetric* tensor, whose value is impossible to store in the left-hand-side object `t1`, which is an antisymmetric tensor. For this reason, the only accumulation operators currently defined for `Tensor` types are `operator+=()` and `operator-=()`, which do not change the symmetry. A consequence of this is that the only reduction operator acting on `Array`s of `Tensor` elements which works is the `sum()` reduction.

Using Multiple Patches

Our next Laplace equation solver uses the class `MultiPatch` to help POOMA take advantage of whatever parallelism is available. An array with a `MultiPatch` engine breaks the total domain into a series of blocks. Such an array is defined as follows:

```
// Define the total domain.
Interval<2> totalDomain(100, 100);

// Define the sizes of the patches (in this case 10×10).
Loc<2> patches(10, 10);

// Create a UniformGridLayout.
UniformGridLayout<2> layout(totalDomain, patches);

// Create the array containing 100 Brick patches, each 10×10.
Array< 2, double, MultiPatch<UniformTag,Brick> > A(layout);
```

The `Interval` declaration specifies the total logical domain of the array being created. The 10×10 `Loc` is then used in the `UniformGridLayout` declaration to specify that the total domain is to be managed using a total of 100 patches. When the `Array` `a` is finally declared, `Array`'s third template parameter is explicitly instantiated using `MultiPatch`, and the layout object `layout` is used as a constructor parameter.

Once all of this has been done, `A` can be used like any other array. However, if a data-parallel expression uses multi-patch arrays, POOMA's evaluator automatically computes values for the patches in parallel. This means that the relaxation program shown below (included in the release as `examples/Solvers/UMPJacobi`) would be able to take full advantage of multiple processors, if the machine it was being run on had them, but would be equally efficient on a

conventional uniprocessor:

```
01 #include "Pooma/Arrays.h"
02
03 #include <iostream>
04
05 const int N = 18; // The size of each side of the domain.
06
07 template<class T1, class E1, class T2, class E2>
08 void
09 ApplyJacobi(
10     const Array<2, T1, E1>      & V, // to be relaxed
11     const ConstArray<2, T2, E2> & b, // fixed term
12     const Range<2>              & IJ // region of calculation
13 ){
14     V(IJ) = 0.25 * (V(IJ+Loc<2>(1,0)) + V(IJ+Loc<2>(-1,0)) +
15                     V(IJ+Loc<2>(0,1)) + V(IJ+Loc<2>(0,-1)) - b(IJ));
16 }
17
18 template<class T1, class E1>
19 void
20 ApplyPeriodic(
21     const Array<2, T1, E1>      & V  // to be wrapped
22 ){
23     // Get the horizontal and vertical extents of the domain.
24     Interval<1> I = V.domain()[0],
25                 J = V.domain()[1];
26
27     // Copy each of the four slices in turn.
28     V(I.first(), J) = V(I.last()-1, J);
29     V(I.last(),  J) = V(I.first()+1,J);
30     V(I, J.first()) = V(I, J.last()-1);
31     V(I, J.last())  = V(I, J.first()+1);
32 }
33
34 int main(
35     int                 argc,           // argument count
36     char *              argv[]          // argument vector
37 ){
38     // Initialize POOMA.
39     Pooma::initialize(argc,argv);
40
41     // The domain with guard cells on the boundary.
42     Interval<2> guarded( Interval<1>(0, N+1), Interval<1>(0, N+1) );
43
44     // Create the layouts.
45     UniformGridLayout<2> guardedLayout( guarded, Loc<2>(4, 4) );
46
47     // The array we'll be solving for.
48     Array<2, double, MultiPatch<UniformTag,Brick> > V(guardedLayout);
49     V = 0.0;
50
51     // The right hand side of the equation.
52     Array<2, double, MultiPatch<UniformTag,Brick> > b(guardedLayout);
53     b = 0.0;
54
55     // Must block since we're doing some scalar code here (see Tutorial 4).
```

```
56      Pooma::blockAndEvaluate();
57      b(3*N/4,   N/4) = -1.0;
58      b(  N/4, 3*N/4) =  1.0;
59
60      // The interior domain, now with stride 2.
61      Range<2> IJ( Range<1>(1, N-1, 2), Range<1>(1, N-1, 2) );
62
63      // Iterate 200 times.
64      for (int i=0; i<200; ++i)
65      {
66          ApplyJacobi(V, b, IJ);
67          ApplyJacobi(V, b, IJ+Loc<2>(1,1));
68          ApplyJacobi(V, b, IJ+Loc<2>(1,0));
69          ApplyJacobi(V, b, IJ+Loc<2>(0,1));
70          ApplyPeriodic(V);
71      }
72
73      // Print out the result.
74      std::cout << V << std::endl;
75
76      // Clean up and report success.
77      Pooma::finalize();
78      return 0;
79  }
```

A program can go one step further, and take advantage of the fact that `MultiPatch` is itself templated. The first template parameter, `LayoutTag`, specifies the type of domain decomposition that is done. If `UniformTag` is specified, then all of the blocks are assumed to have the same size. If `GridTag` is specified, then the domain decomposition can consist of an array of non-uniform blocks, still arranged in a `Dim` dimensional grid (see `examples/Solvers/GMPGuardedJacobi`). Future releases will include a tile-layout that can cover the domain with blocks that are not necessarily arranged on a grid.

The second template parameter specifies the type of `Engine` used in the patches. If `MultiPatch<UniformTag, CompressibleBrick>` is used as a template parameter in an `Array` declaration, then POOMA will not actually allocate memory for a patch if all of the values in that patch are known to be the same. For example, if a wave propagation program initializes all but a few array elements to zero, then the patches whose elements are all zero will be expanded automatically on demand. This can save significant execution time in the early stages of such calculations.

Note that POOMA can deal with `MultiPatch` arrays having different layouts. However, best performance is obtained when all layouts in an expression are the same (though some may have guard layers, as discussed in the following section).

Another variation on this program that uses threads explicitly is presented in the appendix. This program is more complex than the one above, but also has tighter control over what happens and when it happens.

Guard Layers

Multipatch arrays do present a complication to the evaluation of expressions. Evaluation of stencils such as those involved in the Jacobi iteration becomes tricky near the edge of a patch since data will be require from a neighboring patch. This is handled by evaluating the strips near the edge separately from the bulk of the patch. As the overhead for evaluating a patch is roughly constant, small sub-patch evaluations hurt efficiency.

One mechanism for fixing this problem is to introduce *guard* (or ghost) layers. This done by having the individual patches overlap slightly. Each patch still "owns" the same data as before, but surrounds that data with a layer of guards. These guards duplicate data that is owned by other patches, and can only be read from, not written. Now the evaluator can be written as a single loop over the entire owned portion of the patch, with the stencil terms reading from the guard layers. POOMA takes care of keeping the data in the guard layers in sync with the neighboring patches.

The guards described above are known as internal guards. POOMA also supports the notion of external guards. For `Array` objects, external guards are simply syntactic sugar for declaring a layer of cells around the domain of interest. POOMA

`Field` objects hide the external guards and use them to calculate boundary conditions.

One can modify the Jacobi example simply by passing two `GuardLayers` objects to the layout constructor, one specifying the internal guards and another specifying the external guards:

```
// Specify the internal guards
GuardLayers<2> igls(1);

// Specify no external guards
GuardLayers<2> egls(0);

// Define the number of the patches.
Loc<2> patches(4, 4);

// Create a UniformGridLayout with internal guards.
UniformGridLayout<2> guardedLayout( guarded, patches, igls, egls );
```

Complete examples using guard cells are presented in the `UMPGuardedJacobi` and `GMPGuardedJacobi` examples in `examples/Solvers`.

POOMA can support different guard layers for each axis, and for both the high and low faces along each axis. These are specified by initializing the `GuardLayers` object with two raw `int` arrays, such as:

```
int lower[] = { 2, 0, 1 };
int upper[] = { 0, 0, 1 };

GuardLayers<3> internal(lower, upper);
```

This code fragment initialize `internal` to have a single guard layer on the lower side of the first dimension, and one on each the upper and lower sides of the third dimension.

Taking Layout Into Account

We now examine how to construct a loop-based reduction engine that takes into account some of the different ways POOMA arrays can be laid out in memory. Some aspects of this example are left unexplained, or glossed over, since the main intent of this example is to show how intermediate or advanced users of the library can tailor it to their needs.

The most common array layout in POOMA is called a brick layout, and is signaled by the use of the class `Brick` as an engine specifier in template instantiation. Conceptually, a brick is a dense, rectangular patch of multi-dimensional space, such as the area [0..10]×[0..10]. Programs written by the typical user access the elements of bricks using nested loops, the indices of which sweep through the brick's extent along a particular axis. Programs written by POOMA's developers use more complicated access loops in order to take full advantage of cache behavior.

The three functions `accumulateWithLoop()` defined below are the guts of the general-purpose adding routine that we will build up in this example. Each routine loops over the axes of an array of different dimension; the C++ compiler knows which version of the overloaded function to instantiate by pattern-matching the actual dimension of the array being summed with the dimension value specified as the first argument to `ConstArray` (i.e., 1, 2 or 3). The real version of this code has seven versions of `accumulateWithLoop()`, since POOMA arrays can have up to seven dimensions. Note that these functions have to be written explicitly, since there is no way in C++ to create entirely new statements (such as new nested loops) through template expansion.

```
template<class T, class E>
inline T accumulateWithLoop(
    const ConstArray<1,T,E>& x
){
    T sum = 0;
    int f0 = x.first(0), l0 = x.last(0);
    for (int i0=f0; i0<=l0; ++i0)
```

```
        sum += x(i0);
    return sum;
}

template<class T, class E>
inline T accumulateWithLoop(
    const ConstArray<2,T,E>& x
){
    T sum = 0;
    int f0 = x.first(0), f1 = x.first(1);
    int l0 = x.last(0),  l1 = x.last(1);
    for (int i1=f1; i1<=l1; ++i1)
        for (int i0=f0; i0<=l0; ++i0)
            sum += x(i0, i1);
    return sum;
}

template<class T, class E>
inline T accumulateWithLoop(
    const ConstArray<3,T,E>& x
){
    T sum = 0;
    int f0 = x.first(0), f1 = x.first(1), f2 = x.first(2);
    int l0 = x.last(0),  l1 = x.last(1),  l2 = x.last(2);
    for (int i2=f2; i2<=l2; ++i2)
        for (int i1=f1; i1<=l1; ++i1)
            for (int i0=f0; i0<=l0; ++i0)
                sum += x(i0, i1, i2);
    return sum;
}
```

The next step is to write four versions of the interface function that will actually be called by users. These four functions appear the same from a user's point of view (i.e., the syntax that a programmer types in to invoke these functions is indistinguishable). The first version uses explicit specialization to pattern-match arrays that have actual `Brick` engines:

```
template<int D, class T>
T accumulate(
    const ConstArray<D,T,Brick>& x
){
    return accumulateWithLoop(x);
}
```

This function just calls through to whichever version of `accumulateWithLoop()` handles arrays of dimension `D`. Since `accumulateWithLoop()` is an `inline` function, this one extra function call will be eliminated by the compiler when this code is optimized.

The second version of this function handles arrays whose engines are `BrickView`s, rather than `Brick`s. Recall that a `BrickView` is an alias for a subset of the elements in an actual `Brick`. The template class `BrickView` takes a dimension and a Boolean as template arguments; the Boolean specifies whether the `BrickView` can assume a unit stride along its first dimension. Taking a view of a `Brick` leads to this parameter being true; otherwise, it is false.

```
template<int D1, class T, int D2, bool S>
T accumulate(
    const ConstArray< D1, T, BrickView<D2,S> >& x
){
    return accumulateWithLoop(x);
}
```

The third version of `accumulate()` is the one that makes this example interesting:

```
template<int D, class T>
T accumulate(
    const ConstArray< D, T, MultiPatch<UniformTag,Brick> >& x
){
    typename UniformGridLayout<2>::iterator
        i = x.message(GetGridLayoutTag<2>()).begin(),
        e = x.message(GetGridLayoutTag<2>()).end();
    T sum = 0;
    while (i != e)
    {
        sum += accumulate(x(*i));
        ++i;
    }
    return sum;
}
```

Instances of the class `UniformGridLayout` store information about the patches that make up a uniform multi-patch. (They do other things as well; please see the POOMA documentation for the full list.) The first three lines of the function shown above declare a pair of iterators, which the function then uses to iterate through the patches of the array. The expression `x(*i)` accesses a single patch; this patch is then passed to whichever version of `accumulate()` is appropriate for patches of that kind.

Our final version of `accumulate()` exists to ensure that arrays using other storage mechanisms can still be summed. When the C++ compiler expands templates, it takes a more-specific match in preference to a less-specific match. Thus, since `class E` (i.e., a class variable) is used as the third parameter in the template parameter list below, instead of a concrete engine tag class such as `Brick`, the compiler will only choose this version of `accumulate()` when no other version will do:

```
template<int D, class T, class E>
T accumulate(
    const ConstArray<D,T,E>& x
){
    return accumulateWithLoop(x);
}
```

It is important to note that if the specialized versions of `accumulate()` had not been defined, this generic version would return the correct answer for any kind of array storage, including `MultiPatch`. The only advantage of looping over patches explicitly is that it yields better cache usage, and hence higher performance, since patch sizes are usually chosen so that the whole of each patch will fit into cache at once. POOMA therefore allows programmers to make sure that their code is working correctly before they start tuning it, and to tune programs incrementally based on the results of profiling.

For an example of a program that uses ideas like these, but manages threads explicitly, see the appendix.

Component Views

It is often useful to create an array of a structured type, such as a `Vector<3>`, and then select a view consisting of corresponding elements of that type, such as the *Z* component of the position that each `Vector<3>` represents. Such *component views* are closely related conceptually to the component forwarding introduced earlier. POOMA allows programs to create such views where the array type is itself a POOMA type. For example, suppose a program contains the following statements:

```
Array<2, Vector<3> > a(10, 10);
Array<2> b(10, 10);
b = a.comp(2);
```

The right-hand side of the assignment statement is a view of the third components of all of `a`'s vector elements. This is

implemented as an `Array` whose engine is a component forwarding engine. Data is only accessed on demand: the expression `a.comp(2)` does *not* copy values out of `a` into temporary storage.

If the source array of a component view is writable (i.e. not a `ConstArray`), then that component view can appear on either side of the assignment operator. For example:

```
Interval<1> I(5);
a(2, I).comp(1) = 3.14;
```

sets the second component of all of the vector elements in the slice to 3.14. The class `ComponentView` can also be used to make an object to store the view, as in:

```
ComponentView<Loc<1>, Array<2, Vector<3> > > va = a.comp(1);
```

Here, the argument "Loc<1>" indicates that the component is singly-indexed. Up to 7 indices are supported, since programs can make `Array`s with `Array` elements.

Summary

POOMA does its best to insulate programmers from the details of parallelism and modern memory hierarchies, but eventually these issues must be dealt with if high performance is to be achieved. This tutorial has therefore introduced some of the characteristics and capabilities of the POOMA library which developers must take into account in order to get the best possible performance from modern parallel computers, and some of the techniques (such as traits classes) which are used to implement the library.

[Prev] [Home] [Next]

POOMA Tutorial 6
Indirect Addressing

Contents:

Introduction
Notation
Example
Summary

Introduction

Indirect addressing is a fundamental operation in many numerical and scientific algorithms. Instead of accessing array elements with loop indices directly, indirect addressing uses the element of one array (sometimes called an index table or indirection table) as indices for another array. These index tables can store either static information (such as the neighbors of points in an unstructured mesh), or dynamic information (such as the sorting order of the elements in a vector).

This tutorial shows how to perform indirect addressing in POOMA, and discusses some of the subtleties and complexities that arise when indirection and multithreading are combined.

Notation

Suppose that the array `X` contains the following values:

a	b	c	d

while the array `J` contains:

3	0	1	2

A program could re-order the elements of `X` while copying them into another array `Y` using the following loop:

```
for (int i=0; i<4; ++i)
{
  Y = X(J(i));
}
```

The effect of this would be to fill `Y` with the following values:

d	a	b	c

POOMA allows this operation to be expressed more economically, simply by using the array `J` as a subscript on the array `X` directly:

```
Y = X(J);
```

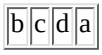
Indirect addressing on the source side of an assignment is sometimes called "pull" addressing, since the index array's values are being used to "pull" values from the source into the destination. POOMA also supports "push" addressing, in which the index array is used on the destination side of the assignment. The syntax for this is simply:

```
Y(J) = X;
```

which is equivalent to the loop:

```
for (int i=0; i<4; ++i)
{
  Y(J(i)) = X;
}
```

This operation would fill `Y` with:



So long as `J` is a strict permutation of the indices *0...N-1*, this will have the same effect as the loop shown above. The effects of this statement if `J` has repeated or missing elements are discussed in the next section.

One-dimensional arrays of integers can be used as subscripts to one-dimensional arrays of any other type, but how are multi-dimensional arrays to be subscripted? In POOMA, the answer is to use arrays whose elements are of type `Loc`. An `Array<Loc<2>>`, for example, can be used to re-order the elements of a two-dimensional array, since each element of the index array can act as a coordinate in the data array. Similarly, an `Array<Loc<3>>` can be used to subscript a 3-dimensional array of any type. Future releases of POOMA will support higher-dimensional indirect addressing as well.

Indirect addressing is a very powerful tool, but must be used carefully. The most important consideration is that the order of data movement during indirection is not defined. If indirection is performed using an index table that sends many values to a single location, for example, then there is no way to predict which of those values will be written into that location.

However, indirect addressing can always be used to *read* values safely, and to thereby perform a scatter operation. Suppose a source array `S` contains the values `[3.14, 2.71]`, while an index array `IA` contains the values [0,1,0,0,1,1]. The expression `S(IA)` yields:

```
[3.14, 2.71, 3.14, 3.14, 2.71, 2.71]
```

and can always be used safely on the right-hand side of an expression. This works because the domain of `a(b)` is the domain of `b`. In expressions like `a(b) = c`, the domains of `c` and `b` have to match, but the domain of `a` can be arbitrary.

Example

The example for this tutorial is a 1-dimensional Fast Fourier Transform (FFT) that shuffles data using indirect addressing. This FFT implementation is not efficient---it recalculates trigonometric constants repeatedly, for example, rather than pre-calculating and storing them---but it does illustrate the power of indirection.

The source for this example is included in the release in the file `examples/Indirection/FFT/FFT.cpp`. The main body of this program initializes POOMA, creates and fills an array of complex values, transforms it, and prints the result of that transform, as shown below:

```
138  int main(int argc, char* argv[])
139  {
140    Pooma::initialize(argc, argv);
141
142    int size = 16;
143
144    Array<1, complex<double>, Brick> a(size);
145
146    int i;
147    for (i = 0; i < size; ++i)
148    {
149      a(i) = sin(4*i*Pi/size);
150    }
151
152    std::cout << a << std::endl;
153
154    fft(a);
155
156    std::cout << a << std::endl;
```

```
157
158    Pooma::finalize();
159    return 0;
160  }
```

The key statement is on line 154, where the `fft()` function is invoked. The program contains two overloaded versions of this function. The first version, shown below, determines the level of the FFT (i.e. the number of recursive steps the calculation requires), then invokes the second version, which actually performs the calculations. (Note that in this simple example, the input array's length is required to be a power of two. Also, `Pi` is defined as a `static const double` equal to the value of pi computed from the expression `acos(-1.0)` because some compilers do not define mathematical constants such as `M_PI` in the <math.h> header file.)

```
117  void fft(const Array<1, complex<double>, Brick> &array)
118  {
119    int size = array.domain().size();
120
121    // Determine size as power of 2
122    int level = -1;
123    while (size > 1)
124    {
125      PAssert(!(size & 1));
126      ++level;
127      size /= 2;
128    }
129
130    if (level >= 0)
131      fft(array, level);
132  }
```

The second version of `fft()` does the real number-crunching. If the computation has reached its final stage, odd and even elements are combined directly (lines 106-111). If the computation is still recursing, the elements are shuffled, a half-sized transform is applied on each subsection, and the results are combined (lines 100-102). All of these operations use indirect addressing to move data values around. Most of the rest of the program can be viewed as infrastructure needed to make this data movement simple and efficient.

```
083  void fft(const Array<1,complex<double>,Brick> &array, int level)
084  {
085    Interval<1> domain = array.domain();
086
087    if (level > 0)
088    {
089      ConstArray<1, int,             IndexFunction<LeftMap> >    left(domain);
090      ConstArray<1, int,             IndexFunction<RightMap> >   right(domain);
091      ConstArray<1, int,             IndexFunction<ShuffleMap> > shuffle(domain);
092      ConstArray<1, complex<double>, IndexFunction<TrigFactor> > trig(domain);
093
094      left.engine().setFunctor(LeftMap(level));
095      right.engine().setFunctor(RightMap(level));
096      shuffle.engine().setFunctor(ShuffleMap(level));
097      trig.engine().setFunctor(TrigFactor(level));
098
099      // Shuffle values, compute n/2 length ffts, combine results.
100      array = array(shuffle);
101      fft(array, level-1);
102      array = array(left) + trig*array(right);
103    }
104    else
105    {
106      int size = domain.size();
107      Range<1> left (0, size-2, 2),
108               right(1, size-1, 2);
```

```
109
110       array(left) += array(right);
111       array(right) = array(left) - 2.0 * array(right);
112     }
113   }
```

The shuffling step on line 100 uses an indirection array called `shuffle` to pull values into the right positions. This array, which is declared on line 91, is a `ConstArray` of integers. Instead of storing the values, however, the array calculates them on the fly using an `IndexFunction` engine, which is bound to the array on line 96. The `IndexFunction` engine works as one would expect: having been specialized with a user-defined class with an overloaded `operator()`, the engine transforms an index `i` into some new value by calling that `operator()`. In this case, the specializing class is `ShuffleMap`, which is shown below:

```
003   struct ShuffleMap
004   {
005     ShuffleMap(int n = 0)
006       : degree_m(n)
007     {
008       nbit_m = 1 << n;
009       mask1_m = nbit_m - 1;
010       mask2_m = ~(nbit_m | mask1_m);
011     }
012
013     int operator()(int i) const
014     {
015       return
016         (mask2_m & i)
017         | ( (mask1_m & i) << 1 )
018         | ( (nbit_m & i) ? 1 : 0 );
019     }
020
021     int nbit_m, mask1_m, mask2_m, degree_m;
022   };
```

Similar engines are used to select and combine elements of the arrays after the sub-FFTs have been performed. These use the overloaded `operator()` in the classes `LeftMap` and `RightMap`, shown below:

```
029   struct LeftMap
030   {
031     LeftMap(int n = 0)
032       : nbit_m(~(1 << n))
033     { }
034
035     int operator()(int i) const
036     {
037       return (nbit_m & i);
038     }
039
040     int nbit_m;
041   };
042
043   struct RightMap
044   {
045     RightMap(int n = 0)
046       : nbit_m(1 << n)
047     { }
048
049     int operator()(int i) const
050     {
051       return (nbit_m | i);
052     }
053
```

```
054    int nbit_m;
055  };
```

Finally, an `IndexFunction` engine is also used to calculate the trigonometric weights used in combining. This `IndexFunction` is an extreme example of trading time for space: it does not store anything, but repeatedly recalculates factors on demand.

```
065  struct TrigFactor
066  {
067    TrigFactor(int n = 0)
068      : n_m(1 << n)
069    { }
070
071    complex<double> operator()(int i) const
072    {
073      return complex<double>(cos(Pi*i/n_m), sin(Pi*i/n_m));
074    }
075
076    int n_m;
077  };
```

Summary

Efficient support for indirect addressing---the use of the values in one array to change the way another array's elements are accessed---is one of the features that characterizes full-strength numerical libraries. This release of POOMA supports indirect addressing in both "push" and "pull" modes using conventional data-parallel syntax, without compromising the performance of regular index operations.

[Prev] [Home] [Next]

POOMA Tutorial 7
Meshes, Centerings, Geometries, and Fields

Contents:

Introduction
Overview
Mesh
Centering
  A Note on Allocation
Geometry
  A Note on Positions
Field
  A Note on Allocation
Example: One-Dimensional Scalar Advection
Example: N-Dimensional Scalar Advection
Summary

Introduction

As mentioned in an earlier tutorial, POOMA provides classes that know enough about their own spatial structure to manage stencil operations and the like automatically. The most important of these classes, `Field`, is the main subject of this tutorial. In order to understand how discrete `Field`s are built and used, however, it is necessary to understand how meshes are represented, what a centering is, and how `DiscreteGeometry` and related classes are used. After a quick overview of how these concepts tie together, this tutorial therefore describes POOMA's mesh abstraction, then its representation of centerings, then its geometry abstraction, and finally how the two are tied together by `Field`.

Overview

An array is a multi-element data structure, each of whose elements is specified by one or more indices. An array's indices don't mean anything in and of themselves; their only purpose is to order the array's elements.

A field, on the other hand, defines a set of values on a region of space. As with an array, the indices used to access a field's elements specify ordering and adjacency. Unlike an array's indices, however, a field's indices also have meaning: there is no "place" corresponding to element (2,2) of an array (except in a very abstract sense), but the third element of the third row of a field has some definite position in space.

In order to specify a field, a library such as POOMA must specify the locations at which the field's values are defined, and describe what happens at the boundaries of that region in space. The first of these tasks is handled in POOMA by geometry classes, which are used as template parameters to the `Field` class. This release of POOMA only provides geometry classes for discrete `Field`s; geometry classes capable of representing continuous `Field`s may be included in future releases. This release of POOMA further restricts all of its predefined geometry classes to represent discrete sets of points defined relative to a *mesh*, which is a set of connected points that spans a region of space. Meshes are discussed in the next section.

In addition to a mesh type, the geometry classes are parameterized by a *centering* type which describes the relationship of the geometry's points to the mesh. As discussed below, the points' locations relative to the mesh can, for example, be the mesh vertices, the cell centers, the face centers, or the edge centers. POOMA provides several classes to represent the mesh abstraction, several classes to represent the centering abstraction, and a `DiscreteGeometry<Centering,Mesh>` class which combines these to represent geometries. These are all described in detail later.

The second task of fields---describing what happens at the boundaries of a region of space---is handled in POOMA by boundary condition classes. So far, POOMA provides only predefined boundary condition classes for discrete Fields centered relative to logically rectilinear meshes. Various kinds of reflecting, constant, extrapolating, and periodic boundary conditions are supported.

Geometry and boundary-condition classes support the application-level `Field` class, which represents the field abstraction. Like an `Array`, a `Field` can be used in data-parallel expressions, subscripted with scalar indices and domains of various shapes and sizes, and so on. However, `Field`s also have an understanding of the spatial locations of their values, and of their boundary conditions. For example, the spatial locations of a `Field`'s elements can be accessed using the member function `Field::x()`.

Mesh

A *mesh* is a discrete domain (i.e. a discrete set of points in coordinate space) and some kind of connectivity rule. This rule specifies which points in the mesh are connected to which others to form *edges*. In turn, sets of edges define *faces*, and sets of faces specify the boundaries of *zones* or *cells* in space.

POOMA contains a set of related classes to represent meshes. The classes in this release represent meshes which are logically rectilinear. They are not necessarily physically rectilinear because they support curvilinear as well as Cartesian coordinates. However, the template arguments, constructor parameters, and accessor methods of these classes allow for future releases to provide more general meshes, such as unstructured meshes with heterogeneous zones.

Like most POOMA classes, meshes can be constructed and initialized in two ways. The first technique is to pass parameters to a constructor to initialize the mesh's characteristics. The second is to construct the mesh using its default constructor, and then call its `initialize()` method with the parameters that would have been passed to a more complex constructor. (This second technique is typically used when allocating arrays of meshes.) All of POOMA's mesh classes provide a method called `initialized()`, which only returns `true` if the object has been fully initialized.

`UniformRectilinearMesh` is the simplest of POOMA's mesh classes. It represents a region of space that is divided at regular intervals along each axis (although the spacing along different axes may be different). In the 3-dimensional case, for example, the faces of a `UniformRectilinearMesh` are rectangles. Each zone is a block with six faces, and is $dx \times dy \times dz$ in size, where dx, dy, and dz are the spacings along the mesh's three axes.

The `RectilinearMesh` class generalizes `UniformRectilinearMesh` by allowing the spacings to vary along each axis. This kind of mesh is sometimes called a Cartesian-product or tensor-product mesh. The divisions along each axis a^i are defined by a set of intervals $da^i = \{da^i_0, da^i_1, ..., da^i_N\}$ (so that the j^{th} interval on axis i has width da^i_j). The whole mesh is then defined by the outer product $da^0 \times da^1 \times ... \times da^{R-1}$ (where R is the rank of the mesh, i.e. the number of dimensions it has).

The template parameters for `RectilinearMesh` and `UniformRectilinearMesh` are identical, and both support the same basic set of constructors. The main difference between the two classes from a user's point of view is the extra constructors that `RectilinearMesh` provides. For clarity's sake, only the the two-dimensional constructors are shown below. Both classes define constructors which specify defaults for the origin and spacing; more constructors may be added in future releases.

```
template<int Dim,
         class CoordinateSystem = Cartesian<Dim>,
         class T = POOMA_DEFAULT_POSITION_TYPE>
class UniformRectilinearMesh
{
  template<class Dom1, class Dom2>
  UniformRectilinearMesh(const Dom1 &d1,
                         const Dom2 &d2,
                         PointType_t origin,
                         PointType_t spacings)
  {
    constructor body
  }

  rest of class
};

template<int Dim,
         class CoordinateSystem = Cartesian<Dim>,
         class T = POOMA_DEFAULT_POSITION_TYPE>
```

```
class RectilinearMesh
{
  template<class Dom1, class Dom2, class EngineTag>
  RectilinearMesh(const Dom1 &d1,
                  const Dom2 &d2,
                  const PointType_t &origin,
                  const Vector<Dim, Array<1, T, EngineTag > > &spacings,
                  const Vector<2*Dim, MeshBC> &mbc)
  {
    constructor body
  }

  rest of class
};
```

The $\texttt{Dom}_i$ arguments to the constructors must be domains, and serve the same purpose as the domain constructor arguments used by the `Array` class. Only one argument is needed if that argument is a `Dim`-dimensional domain such as an `Interval<Dim>`. However, unlike `Array`s, the domain must be zero-based, i.e. the origin of its index space must be *[0,0,...0]*. (This requirement may be relaxed in future versions of POOMA.) The spatial origin of each type of grid is specified by the constructor's `origin` parameter. `UniformRectilinearMesh` then takes another point, `spacings`, whose values specify the spacings along each axis.

The inter-element spacings for a `RectilinerMesh`, on the other hand, are specified using a `Vector` of one-dimensional `Array`s. Such a structure can be defined and filled using code like the following:

```
Vector<D, Array<1,int> > spacings;
for (d = 0; d < D; d++) {
  spacings(d).initialize(cellDomain[d]);
  for (i = 0; i < cellDomain[d].size(); i++) {
    spacings(d)(i) = (i+1)*10;
  }
}
```

For `RectilinearMesh`, the mesh's boundary conditions are specified by giving an enumeration element for each face of the mesh (which is why there are twice as many boundary condition specifiers as mesh dimensions). The values allowed for the mesh boundary conditions in this release of POOMA, which are defined in `src/Meshes/MeshBC.h`, are `LinearExtrapolate`, `Periodic`, and `Reflective`. For `UniformRectilinearMesh`, the only sensible boundary condition is linear extrapolation (extension using the constant spacings below the origin and beyond the physical mesh upper boundary), which is built into the class; its constructors do not include `MeshBC` enum parameters.

Of course, before a set of points in space or spacings between them can sensibly be specified, a coordinate system must be chosen. This is the purpose of the `CoordinateSystem` template parameter. Its default value, `Cartesian`, produces a Cartesian (truly rectilinear) mesh. Other coordinate systems can also be used: `Cylindrical`, for example, produces a cylindrical coordinate system which is curvilinear. The discrete mesh, however, is indexable like a Cartesian mesh, i.e. it is still logically rectilinear.

In order to allow applications to operate on meshes without hard-coding the mesh's size, spacing, or coordinate system, the mesh classes store information about their domains in `Array` data members. (Where possible, these arrays are implemented using compute engines, so that memory is not wasted recording simple sequences of values.) Once accessed, these information arrays can be used in data-parallel expressions like any others. In particular, they are often used with stencils to implement differential operators such as `div()`and `grad()`(as discussed in the next tutorial).

The `Array`s in POOMA's mesh classes have *guard layers*, which are extra elements outside their calculation domain whose values are defined by the mesh's boundary conditions. All of the mesh classes in this release automatically create *guard layers* that have $N_D/2$ elements along each axis D, where N_D is the number of vertices along that axis. This provides enough space for any plausible accesses to mesh data outside the mesh boundaries, such as locating the nearest vertex to a particle outside the boundary, or implementing a stencil operating on a `Field` centered at the mesh vertices which uses `Field` values at, and mesh spacings between, vertices beyond the boundary by a distance corresponding to the stencil width.

A mesh's positional data can be accessed using two pairs of public methods. `physicalDomain()` returns the mesh's physical domain (i.e. its vertex index domain), excluding its guard vertices. `physicalCellDomain()` returns a domain representing the mesh's cells; for a logically-rectilinear mesh, this is just one element smaller per dimension than `physicalDomain()` (since a rectilinear mesh has one fewer cells than vertices). Similarly, `totalDomain()` returns the domain of the mesh including its guard

vertices, and `totalCellDomain()` performs a similar function for the mesh's cells. The methods `vertexPositions()` and `vertexDeltas()` access the mesh's vertices and spacings respectively. All these methods return references to `Array` data members, which can then be used like any other (read-only) `Array`.

Centering

A mesh does not fully specify the geometry of a discrete field until it is combined with a centering. Centerings are defined relative to the features that uniquely identify a mesh, such as its vertices, zones, faces, and edges. Figure 1 illustrates these features for an example mesh zone in one, two, and three dimensions.

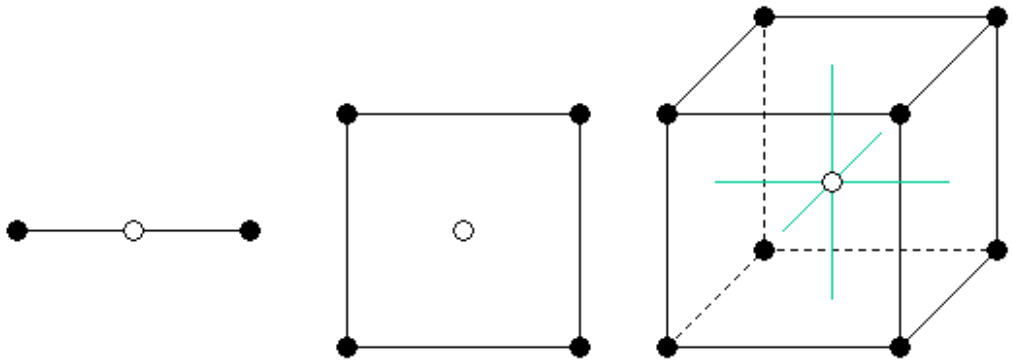


Figure 1: Sample rectilinear mesh zones. Black circles are the vertices, and empty circles are the zone centers. The green axes help show that the zone center in 3D is in the physical center of the rectangular parallelepiped.

The header file `r2/src/Geometry/CenteringTags.h` defines several classes which specify centerings in POOMA when used as template parameters to the `DiscreteGeometry` class discussed in the next section. The first two are non-template classes whose definitions are fairly simple:

```
struct Cell;
struct Vert;
```

For rectilinear meshes , these centering positions are just the white and black circles, respectively, from Figure 1.

The third predefined class in `CenteringTags.h` is a parameterized class specifically designed for logically rectilinear meshes, whose zones, vertices, faces, and edges can all be indexed in multi-dimensional array style (i.e. using `(i,j,k)`-style indices):

```
template <int Dim,
          class RectilinearCenteringTag,
          bool Componentwise = RectilinearCenteringTag::componentwise,
          int TensorRank     = RectilinearCenteringTag::tensorRank,
          int NComponents    = RectilinearCenteringTag::nComponents>
class RectilinearCentering
```

The `RectilinearCenteringTag` template parameter can be instantiated using a class whose centerings which are defined componentwise. This means that each component of a multicomponent field element type such as `Vector` or `Tensor` can have its own independent centering position. The value of the Boolean template parameter `Componentwise` flags whether this is the case: if it is `false`, then all components of each multicomponent `Field` element are centered at the same point, rather than different points.

The `TensorRank` and `NComponents` parameters are required for componentwise centerings. `TensorRank` is the number of indices required to index a component of the multicomponent field element type, i.e. 1 for `Vector`s, and 2 for `Tensor`s. `NComponents` is the number of values indexed by each component index, such as `Dim` for `Vector<Dim>` or `Tensor<Dim>`.

The actual descriptive information about the centering is in the `RectilinearCenteringTag` parameter. POOMA provides a set of classes and class templates that can be used as the `RectilinearCenteringTag` parameter:

```
// Centering on faces perpendicular to Direction:
template<int Direction>
```

```
class FaceRCTag;

// Centering on edges parallel to Direction:
template<int Direction>
class EdgeRCTag;

// Componentwise centering; each component centered on face perpendicular to
// that component's unit-vector direction:
template<int Dim>
class VectorFaceRCTag;

// Componentwise centering; each component centered on face parallel to that
// component's unit-vector direction:
template<int Dim>
class VectorEdgeRCTag;
```

As an example, the `FaceRCTag` in three dimensions defines centering points on the zones' rectangular face centers, perpendicular to the direction specified by the template parameter. With `Direction=0` (the *X* direction), this defines the face centers perpendicular to the *X* axis. In two dimensions, zone faces and zone edges are degenerate; in one dimension, faces are further degenerate with vertices. Figure 2 shows the `FaceRCTag<0>` centering positions (red circles) relative to a single zone in these cases.

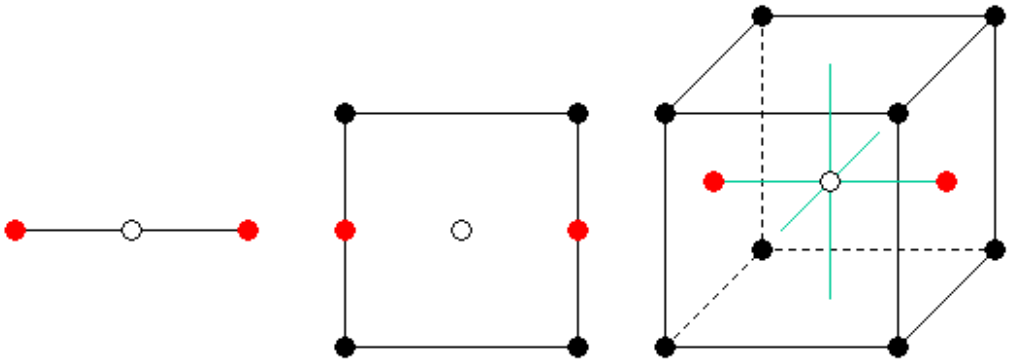


Figure 2: Centering positions of `FaceRCTag<0>` in 1, 2, and 3 dimensions.

Figure 3 shows an example two-dimensional mesh with 4×4 vertices (and thus 3×3 cells), with the complete set of `FaceRCTag<0>` centering points shown. Note that it is really the geometry class using the centering class that determines where the coordinate locations of the centering points are; the figure shows the standard definition (i.e. the geometric centers of the faces). Note also that the number of centering points is equal to the number of cells in the *Y* direction and the number of vertices in the *X* direction. A geometry class using this centering would provide centering position vectors indexable on this physical indexing domain.

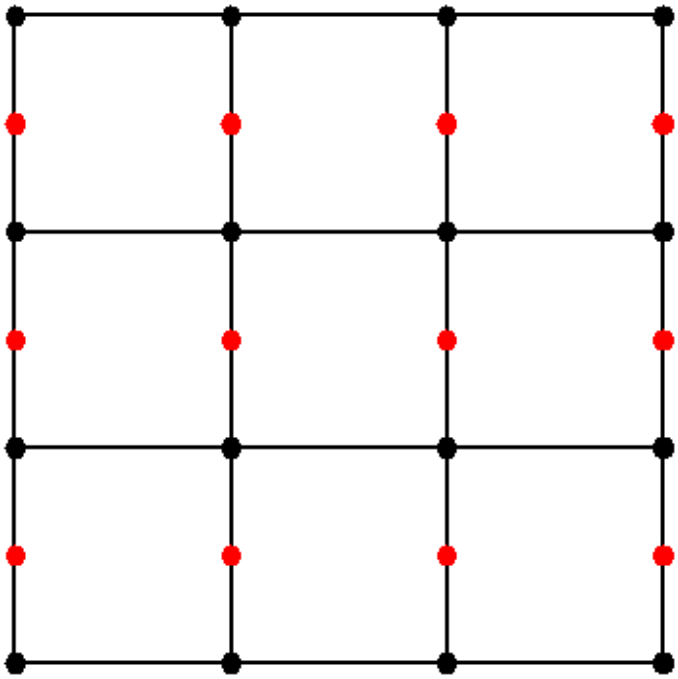


Figure 3: Two-dimensional mesh with complete set of centering points.

As an example of componentwise centering, consider `RectilinearCentering<2,VectorFaceRCTag<2>>`. The *Y* components of a field element of `Vector` type are centered on the faces perpendicular to the *Y* axis, while the *X* components are centered on the faces perpendicular to *X*. Figure 4 illustrates this, by showing the *X* and *Y* components as horizontal and vertical arrows rooted at their centering points. The dotted blue lines indicate which pairs of components are components of a single field element. The green arrows indicate valid *X* and *Y* components at the extremal high-end faces. It is only legal to refer to the one valid component of a vector at this location (using its corresponding *IJK* index). The companion perpendicular components for these values are not defined. (See the note on allocation below for more details.)

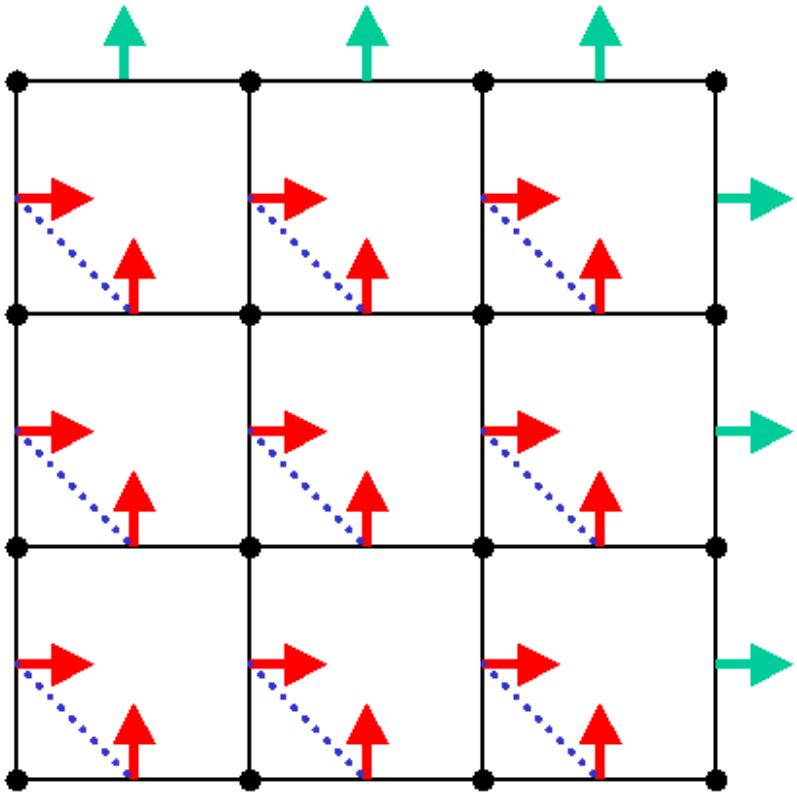


Figure 4: Example of componentwise centering, showing RectilinearCentering<2,VectorFace<2>>

A Note on Allocation

For componentwise rectilinear centerings such as `RectilinearCentering<2,VectorFace<2> >`, POOMA currently allocates `Field` domains (and `Array` domains in the associated `DiscreteGeometry`) with storage for `nVerts` elements in each dimension, so storage for a `Vector` with both components at these extremal locations is allocated, but only the valid component is legally accessible.

Geometry

The next layer of support in POOMA for fields is its geometry abstraction. A geometry is a set of points in a coordinate space. This implies a definition of a coordinate system, an explicit or implicit specification of the points in the set, and what if any boundaries bound the set of points. A geometry might be a continuous set of points, but currently POOMA only provides geometry classes to represent discrete sets of points. Furthermore, POOMA's current geometry classes are restricted to sets of points defined relative to a mesh (represented by one of the POOMA mesh classes described above) according to a centering (represented by one of the POOMA centering classes described above).

Geometries are described in this release of POOMA by partial specializations of the `DiscreteGeometry` class template. `DiscreteGeometry` itself is defined in `src/Geometry/DiscreteGeometry.h`. The class has an empty body (i.e. no methods or data members), and is parameterized as:

```
template<class Centering, class Mesh> class DiscreteGeometry;
```

The two header files `DiscreteGeometry.URM.h` and `DiscreteGeometry.RM.h` instantiate this class with particular template parameters to create the `UniformRectilinearMesh` and `RectilinearMesh` classes respectively. Both of these classes inherit from the `RectilinearGeometryBase` class, which among other things defines default implementations for `DiscreteGeometry`'s `x()`, `totalDomain()`, and `physicalDomain()` methods. `Field` relies on these to implement its own methods---for example, `Field::x()` simply forwards its arguments to its geometry data member, on the assumption that this member will itself have a method called `x()`.

By default, a POOMA geometry does not have any guard cells, i.e. its total domain is the same as its physical domain. (See the section on meshes for an explanation of these terms.) An application can request guard layers for a geometry by passing a `GuardLayers` object to the geometry's constructor, or equivalently its `initialize()` method. `GuardLayers` is defined in `src/Layout/GuardLayers.h`, and simply describes the depth of the guard layer along each axis.

POOMA's geometry abstraction describes a set of points in space, and is intended to serve primarily as a domain (in the functional sense) of something like a field. In order to be used in this way, i.e. in order to be used as the `Geometry` template parameter to POOMA's `Field` class, a class must define certain constants, types, and methods. The two required constants are:

`dimensions`:

The (integer) dimensionality of the set of points (either the dimensionality of the space, or a lower value if the set is a lower-dimensional surface).

`coordinateDimensions`:

The (integer) dimensionality of the coordinate system (i.e. the dimensionality of the space the geometry defines).

The types which a geometry class must define are:

`CoordinateSystem_t`:

The type of the coordinate system.

`Domain_t`:

The type of the geometry's physical and total domains (i.e. the type of the objects used to represent the geometry's set of points). This is also usually obtained from the geometry's underlying mesh.

`PointType_t`:

The type that represents a point in the coordinate space of the geometry.

`PositionArray_t`:

The type of `ConstArray` returned by the `x()` method described below.

`PositionArray_t` is the type of `Array` object "storing" the geometry's set of position values. For the `DiscreteGeometry` types based on rectilinear meshes provided in this release of POOMA (i.e. those whose `Mesh` template parameter is `UniformRectilinearMesh<Dim>` or `RectilinearMesh<Dim>`), `PositionArray_t` is an `Array<Dim,PointType_t,PositionFunctor_t>`. For a continuous geometry, this would be some kind of continuous `Array` type.

The array domain of that `Array` has type `Domain_t`. `Domain_t` must be a type which can serve as a constructor argument for that POOMA `Array`, and must have appropriate dimensionality. For the `DiscreteGeometry` classes mentioned in the previous paragraph, `Domain_t` is a `typedef` for `Interval<Dim>`. For a continuous geometry, it would be some object representing a continuous domain, like a sphere or a spline-surface-bounded solid.

Finally, a class which is to be used as a geometry must define the following methods:

`physicalDomain()`:

Returns this geometry's physical domain, i.e. an instance of some class representing the set of points in the domain's interior, not including its global guard layers. This can be an explicit representation, such as a container of point values, or an implicit representation, such as a parameterized function object defining the bounding surface of the domain, with a method to determine whether a point in the space is inside or outside the set. The type of this object must be `Domain_t`,

`totalDomain()`:

Returns the geometry's total domain (including global guard layers). This method must be implemented even if the geometry has no guard layers; in such a case, it must return the same domain that is returned by `physicalDomain()`.

`x()`:

returns an array of centering positions corresponding to the total domain.

The `DiscreteGeometry`-based classes provided with this release of POOMA actually provide a richer interface than the one described above. First, each of these classes defines the following constant:

`componentCentered`:

`true` if this field has different centerings for each component, and `false` otherwise.

Second, POOMA's `DiscreteGeometry`-based classes create the following convenience typenames:

`Centering_t`:

the centering tag class. This just exports the `Centering` template parameter value.

`GuardLayers_t`:

The type of the object used to represent guard layers for this geometry.

Finally, the classes based on `DiscreteGeometry` define the methods listed below.

`centering()`:

Returns the centering object for this geometry (i.e. an instance of its `Centering` template parameter).

`guardLayers()`:

Returns the `GuardLayers` object for this geometry.

`initialized()`:

Returns `true` if the mesh has been initialized, and `false` otherwise.

`mesh()`:

Returns the mesh relative to which the `DiscreteGeometry` is defined.

`pointIndex()`:

Given a `Vector<Dim,T>` position in the geometry's mesh space, returns the proper `Loc<Dim>` position in the geometry's domain space that is nearby, taking centering into account.

A Note on Positions

The class used as the `Geometry` template parameter for `Field` must provide methods for returning the spatial positions of its points. All of these methods in the geometry classes in this release of POOMA are based on `Array`s of position `Vector`s which use compute engines. As an example, the `DiscreteGeometry<Cell,Mesh_t>` classes define the locations of the zone centers relative to the set of faces that define a zone. For logically rectilinear meshes, this is typically defined as the geometric center of the zone (which is what `DiscreteGeometry<Cell,Mesh_t>` defines it as), but this is not necessarily the case. A user could, for example, define a geometry class which used a `UniformRectilinear` or `Rectilinear` mesh, but which offset the definition of the zone centers from the geometric centers to implement special types of differential operators.

Field

As stated above, the class `Field` represents both a region of space, and a set of values defined on and around that region---a mapping from points in the region to values. This release of POOMA only supports fields with up to three dimensions, although future releases of the library may support higher-dimensional structures.

`Field` has three template parameters. The first, `Geometry`, defines the region of space. The second and third template parameters to `Field` are like those of `Array`: they specify the type `T` of the field's values, and the type of the engine used for storing or evaluating the field's values. The whole definition is therefore:

```
template<class Geometry,
         class T = POOMA_DEFAULT_ELEMENT_TYPE,
         class EngineTag = POOMA_DEFAULT_ENGINE_TYPE>
class Field : parent classes
{
  body
};
```

A `ConstField` class with the same template parameters is also defined, just as a `ConstArray` is defined to accompany `Array`.

A `Field` has a value of type `T` at every point in the spatial domain defined by its geometry class parameter `Geometry`. In this sense, a `Field` is a concrete representation of a function, whose domain is specified by its geometry, and whose range is the set of values the `Field` contains.

A `Field`'s values can be accessed or modified by subscripting the `Field` with scalar indices or an integer-based indexing domain such as an `Interval` (like an `Array`'s values are accessed or modified). As well as storing values, a `Field` can provide information about the space on which it is defined. If `f` is a `Field`, then `f.x()` is a `ConstArray` with the same number of dimensions as `f`, whose elements are the positions at which `f` is defined. In one dimension, `f.x(0)` is therefore the position of one corner of the physical domain the `Field` represents; in functional terms, the field maps the point `f.x(0)` to the value `f(0)`.

As mentioned above, this release of POOMA only supports discrete fields on regularly-spaced points in up to three dimensions. This restriction may be relaxed in future versions; in particular, continuous geometries and fields may also be supported. In this case, `f.x()` would return a continuous `ConstArray`, which would be accessed using floating-point indices, and which would use some analytic or interpolative function to return values.

The discussion of geometry above has implied the possible existence of layers of guard elements lying around discrete fields. These elements are used to implement boundary conditions, so that discrete operators can treat the "interesting" (i.e. interior) elements of `Field`s uniformly. A `Field` can automatically update parts of its domain using boundary condition objects stored in a list. Before being accessed, these boundary condition objects can be queried as to whether the domain they manage needs updating, and then told to update themselves if necessary.

POOMA predefines boundary-condition classes for use with `Field`s that are based on its rectilinear mesh geometry classes. The current release provides periodic, reflecting, constant, and linear-extrapolation boundary condition types; future releases may may include others. More importantly, the required interface for the boundary condition classes is meant to make it easy for users to implement their own special boundary conditions. By following this interface prescription, applications can attach their own boundary conditions to `Field` objects and have them updated automatically, just as the predefined POOMA boundary conditions are updated. The interface allows writing boundary conditions using high-level array-syntax coding. (See the next tutorial for more information on writing boundary conditions in POOMA.)

Operations on `Field`s with global guard layers might need to access `Field::x()` positional values in those guard layers, for example to implement spatially-dependent boundary conditions, or to implement differential operators. Because of this, the geometry classes which `Field` uses must be able to supply positional values beyond the physical centering positions associated with the `Field`s' physical domain. This, in turn, means that the mesh classes used by discrete geometry classes need to return arrays of vertex positions beyond the edge of the actual mesh boundary, from which the geometry can compute the associated cell and face positions at which the `Field` is defined.

As discussed above, POOMA's mesh classes add guard layers to their contained arrays of positions, spacings, and volumes by making use of the fact that the indexing domain of an `Array` can start some number of elements below zero and extend beyond the number of vertices at the other side. The existence of guard layers affects the information that `Field`s provide about the spatial position of their elements. The expression f.x(0) is actually the position of one corner of the total domain of the `Field` `f` only if `f` has no guard layers, since the rule is that the physical domain of a `Field` is always zero-based. This means that in the presence of guard layers the actual corners of the `Field` will have negative indices. However, it is always true that the `Field` maps the point `f.x(0)` to the value `f(0)`.

The number of guard layers in the `DiscreteGeometry` objects is determined by user input on construction (using `GuardLayers<Dim>` objects), and becomes the number of guard layers that the `Field` itself has as well. The `DiscreteGeometry` uses values from the guard layers in the mesh to fill its arrays of centering-point values (which are returned by its `x()` method). The number of guard layers specified for the `DiscreteGeometry`, and hence for any `Field` that is constructed using the `DiscreteGeometry` object, cannot be larger than the *N/2* number of guard layers automatically defined in the `RectilinearMesh` or `UniformRectilinearMesh` object used to construct the `DiscreteGeometry`.

A Note on Allocation

What's going on under the hood when an application makes a `DiscreteGeometry` object with this `VectorFace` type of componentwise centering for its `Centering` parameter? The `DiscreteGeometry::totalDomain()` method returns a domain with an extent of `nVerts×nVerts×nVerts` (in three dimensions). When the application constructs a `Field` using a geometry object as a constructor argument, it uses `DiscreteGeometry::totalDomain()` in order to allocate its own `Array` storage. The geometry classes have internal `Array` data members called `positions_m` which store the position values accessed by `DiscreteGeometry::x()`; in all the existing `DiscreteGeometry` partial specializations, these `Arrays` have compute-based engines, so they don't allocate any storage.

The domains of these `Arrays` must still be specified. In a geometry class which has `VectorFace` for its `Centering` template parameter, these compute-based `Array` data members have their domains set to `nVerts×nVerts×nVerts`. Any `Field` which uses this geometry (whose `Field::x()` method forwards to `Field::geometry()::x()`) will therefore automatically have its domain aligned with that of the geometry.

Example: One-Dimensional Scalar Advection

The example program in `examples/Field/ScalarAdvection1D` illustrates the features of fields introduced so far by simulating advection in one dimension. A later example in this tutorial shows how to generalize this to handle *N* dimensions.

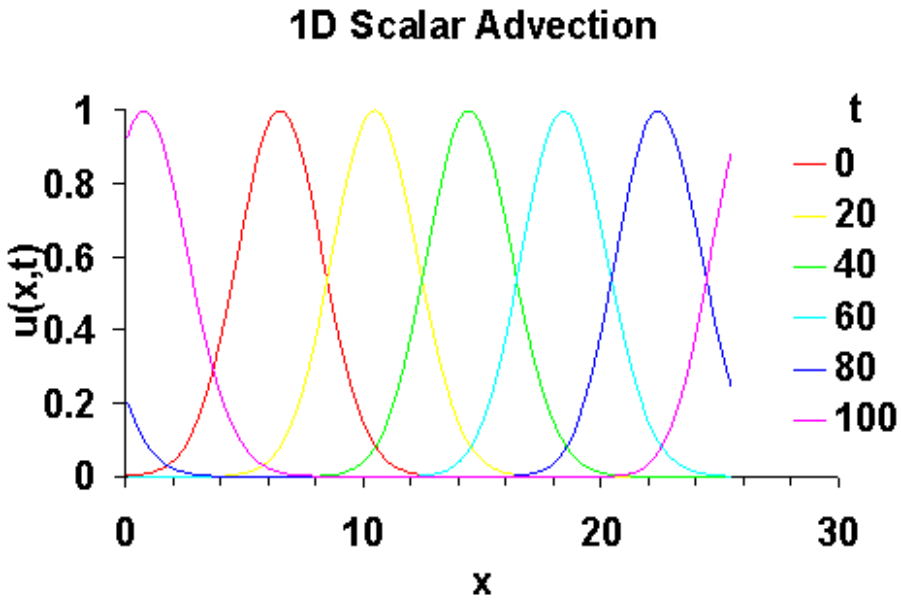
The partial differential equation involved is:

$$du(x,t)/dt = -v * du(x,t)/dx$$

where *v* is a constant propagation speed, and *da/db* represents the partial derivative of *a* with respect to *b*. The analytic solution of this is just a rightwards propagation at speed *v* of the initial condition:

$$u(x,t) = u0(x - vt)$$

The figure below shows that the numerical solution approximates this well.



This equation is a special 1-dimensional version of the general flux-conservative equation:

$$du(x,y,z,t)/dt = - div(F)$$

where F is a vector function:

$$F = (F_x(x,y,z,t), F_y(x,y,z,t), F_z(x,y,z,t))$$

The N-dimensional scalar advection program discussed later solves this equation for the special case where $F_x = v_x*u$, $F_y = v_y*u$, and $F_z = v_z*u$. Note that in one dimension this reduces to exactly the 1D PDE stated above.

The one-dimensional code is shown below. For this particular differential equation, a simple Euler scheme is unstable, so the code uses a leap-frog method based on the difference equation:

$$(u_j^{n+1} - u_j^{n-1}) / (2\ dt) = - v\ (u_{j+1}^{n} - u_{j-1}^{n}) / (2\ dx)$$

This scheme is primed by executing a single Euler step:

$$(u_j^{n+1} - u_j^{n}) / dt = - v\ (u_{j+1}^{n} - u_{j-1}^{n}) / (2\ dx)$$

```
001  #include "Pooma/Fields.h"
002
003  #include <iostream>
004  using namespace std;
005
006  int main(int argc, char *argv[])
007  {
008    Pooma::initialize(argc,argv);
009
010    // Create the physical domains (1D):
011    const int nVerts = 129;
012    const int nCells = nVerts - 1;
013    Interval<1> vertexDomain(nVerts);
014
```

```
015    // Create the (uniform, logically rectilinear) mesh:
016    const Vector<1> origin(0.0), spacings(0.2);
017    typedef UniformRectilinearMesh<1> Mesh_t;
018    Mesh_t mesh(vertexDomain, origin, spacings);
019
020    // Create two geometry objects - one allowing 1 guard layer to
021    // account for stencil width and another with no guard layers to support
022    // temporaries:
023    typedef DiscreteGeometry<Cell, UniformRectilinearMesh<1> > Geometry_t ;
024    Geometry_t geom1c(mesh, GuardLayers<1>(1));
025    Geometry_t geom1ng(mesh);
026
027    // Create the Fields:
028
029    // The flow Field u(x,t):
030    Field<Geometry_t> u(geom1c);
031    // The same, stored at the previous timestep for staggered leapfrog
032    // plus a useful temporary:
033    Field<Geometry_t> uPrev(geom1ng), uTemp(geom1ng);
034
035    // Initialize flow Field to zero everywhere, even global guard layers:
036    u.all() = 0.0;
037
038    // Set up Periodic Face boundary conditions:
039    u.addBoundaryCondition(PeriodicFaceBC(0));     // Low X face
040    u.addBoundaryCondition(PeriodicFaceBC(1));     // High X face
041
042    // Used various places below:
043    Interval<1> pd = u.physicalDomain();
044
045    // Load initial condition u(x,0), a pulse centered around nCells/4 and
046    // decaying to zero away from nCells/4 both directions, with a height of 1.0,
047    // with a half-width of nCells/8:
048    const double pulseWidth = spacings(0)*nCells/8;
049    const double u0 = u.x(nCells/4)(0);
050    u = 1.0*exp(-pow2(u.xComp(0)(pd)-u0)/(2.0*pulseWidth));
051
052    // Output the initial field:
053    std::cout << "Time = 0:\n";
054    std::cout << u << std::endl;
055
056    const double v = 0.2;  // Propagation velocity
057    const double dt = 0.1; // Timestep
058
059    // Prime the leapfrog by setting the field at the previous timestep
060    // using the initial conditions:
061    uPrev = u;
062
063    // Do a preliminary timestep using forward Euler, coded directly using
064    // data-parallel syntax:
065    u -= 0.5*v*dt*(u(pd+1)-u(pd-1))/spacings(0);
066
067    // Now use staggered leapfrog (second-order) for the remainder of the
068    // timesteps:
069    for (int timestep = 2; timestep <= 1000; timestep++)
070    {
071      uTemp = u;
072      u = uPrev-v*dt*(u(pd+1)-u(pd-1))/spacings(0);
073      if ((timestep % 200) == 0)
074      {
075        // Output the field at the current timestep:
```

```
076         std::cout << "Time = " << timestep*dt << ":\n";
077         std::cout << u << std::endl;
078       }
079       uPrev = uTemp;
080     }
081
082     Pooma::finalize();
083     return 0;
084   }
```

After initializing the POOMA library, this code sets up the world on which the equation is to be solved. Lines 11-13 define the size of the simulation, while lines 16-18 define the mesh on which calculations will be performed. Lines 23-25 then use this mesh to define two geometry objects. The first, `geom1c`, includes a guard layer, so that a finite difference stencil can be applied safely. The second, `geom1ng`, does not include this guard layer, but instead only represents the "actual" region of the solution. This geometry is used to define temporaries, as discussed below.

The actual flow field variable `u` is declared on line 30. Since this is the variable to which the full stencil is later applied, it uses the full geometry `geom1c` (the one with the guard layer). The `Field` used to record the previous iteration's results, and a general-purpose temporary, are declared on line 33. These `Field`s use the `geom1ng` geometry, which does not include memory for guard layers. While the memory saved by not having guard layers for temporaries is insignificant in this case, it can be substantial on larger problems, and in more dimensions.

The field `u` is initialized to zero everywhere (even in its guard layers) on line 36, using the method `all()` to get a reference to the whole of the field's data. Periodic boundary conditions are then set on lines 39-40. Line 43 then records the bounds on the problem domain in the `Interval pd`.

The statements on lines 48-50 insert a symmetric pulse into the field. The boundary conditions are applied after this is done to ensure that the field is in a consistent state. The values of the field at this point are then printed out, for later conversion into the graph shown earlier.

The constants controlling the simulation are set on lines 56-57, while the advection calculation itself is initialized on lines 61 and 65. The timestep is 0.1, and the propagation velocity is fixed at 0.2 (both in arbitrary units). After storing the initial state of the field in `uPrev`, so that the loop beginning on line 69 will perform its first iteration correctly, the program calculates the first set of new values for the field directly. Note how the domain of this calculation is defined using the `pd` value that was obtained from the field itself. This idiom helps ensure the consistency of large programs, which many juxtapose many different domains. It also helps make the program more robust in the face of incremental evolution: if the declaration of an important variable (like the `Field u`) is altered, calculations involving that variable reflect those alterations automatically.

The loop on lines 69-80 repeatedly updates the `Field` by invoking the calculation on line 72. The bulk of the code in the loop (lines 73-78) simply outputs the state of the `Field` every 200 iterations, so that a graph showing its evolution can be created later. Finally, the library is shut down, and the program terminated, on lines 82-83.

The most important thing to note about this program is the way in which various calculation domains are declared and combined. As a general rule, only a small number of calculation domains are ever declared from scratch; all others are then derived from these. As a corollary, the extent of calculations on `Field`s are usually determined by interrogating the `Field`, rather than by using long-lived `Range`s or other objects. This helps keep the code correct as it evolves, and is also an important step toward generalizing codes such as this to handle an arbitrary number of dimensions.

Example: N-Dimensional Scalar Advection

The differential equation solved in the preceding example is a special 1-dimensional version of the general flux-conservative equation:

$$du(x,y,z,t)/dt = - div(F)$$

where F is a vector function:

$$F = (F_x(x,y,z,t), F_y(x,y,z,t), F_z(x,y,z,t))$$

The N-dimensional scalar advection program discussed in this tutorial solves this equation for the special case where $F_x = v_x*u$, $F_y = v_y*u$, and $F_z = v_z*u$. Note that in one dimension this reduces to the equation shown in the previous example.

The N-dimensional code shown below revisits the scalar advection code shown earlier, using a less dimension-dependent implementation strategy. Again, since a simple Euler scheme is unstable for this particular differential equation, the code uses a

leap-frog method based on the difference equation:

$$(u_{ijk}^{n+1} - u_{ijk}^{n-1}) / (2\ dt) = -\ div(\mathbf{v}\ u_{ijk}^{n})$$

where:

$$\boldsymbol{v}=v_x\boldsymbol{x}+v_y\boldsymbol{y}+v_z\boldsymbol{z}$$

in three dimensions, and the *div()* difference operator on the right-hand side is centered in space about *(i,j,k)*, so that it involves differences of the form:

$$v_x*(u_{i+1,j,k}^{n} - u_{i-1,j,k}^{n})/dx$$

As described in the next tutorial, this is exactly what POOMA's `div()` function does, so the leap-frog timestepping is implemented using:

```
u = uPrev - 2 div<Cell>(v dt u)
```

This scheme is primed by executing a single Euler step, which also uses POOMA's `div()` function to do the space-centered differencing on the right-hand side:

$$(u_j^{n+1} - u_j^{n}) / dt = -\ div(\mathbf{v}\ u_{ijk}^{n})$$

```
u                = u - div<Cell>(v dt u)
```

As we have seen, all of the important classes in POOMA take the dimension of the problem space as a template parameter. Provided all definitions in the program are made in terms of this parameter, or in terms of types exported from POOMA classes by `typedef`s, applications can move from two to three dimensions simply by changing line 13 in the following source code:

```
001  #include "Pooma/Fields.h"
002
003  #include <iostream>
004
005  int main(int argc, char *argv[])
006  {
007    // Set up the library
008    Pooma::initialize(argc,argv);
009
010    // Create the physical domains:
011
012    // Set the dimensionality:
013    const int Dim    = 2;
014    const int nVerts = 129;
015    const int nCells = nVerts - 1;
016    Interval>Dim> vertexDomain;
017    int d;
018    for (d = 0; d < Dim; d++)
019    {
020      vertexDomain[d] = Interval<1>(nVerts);
021    }
022
023    // Create the (uniform, logically rectilinear) mesh.
024    Vector<Dim> origin(0.0), spacings(0.2);
025    typedef UniformRectilinearMesh<Dim> Mesh_t;
026    Mesh_t mesh(vertexDomain, origin, spacings);
027
028    // Create two geometry objects - one allowing 1 guard layer to account for
029    // stencil width and another with no guard layers to support temporaries:
030    typedef DiscreteGeometry<Cell, UniformRectilinearMesh<Dim> > Geometry_t;
031    Geometry_t geom(mesh, GuardLayers<Dim>(1));
032    Geometry_t geomNG(mesh);
033
034    // Create the Fields:
035
```

```
036    // The flow Field u(x,t):
037    Field<Geometry_t> u(geom);
038    // The same, stored at the previous timestep for staggered leapfrog
039    // plus a useful temporary:
040    Field<Geometry_t> uPrev(geomNG), uTemp(geomNG);
041
042    // Initialize Fields to zero everywhere, even global guard layers:
043    u.all() = 0.0;
044
045    // Set up periodic boundary conditions on all mesh faces:
046    u.addBoundaryConditions(AllPeriodicFaceBC());
047
048    // Load initial condition u(x,0), a symmetric pulse centered around nCells/4
049    // and decaying to zero away from nCells/4 all directions, with a height of
050    // 1.0, with a half-width of nCells/8:
051    const double pulseWidth = spacings(0)*nCells/8;
052    Loc<Dim> pulseCenter;
053    for (d = 0; d < Dim; d++) { pulseCenter[d] = Loc<1>(nCells/4); }
054    Vector<Dim> u0 = u.x(pulseCenter);
055    u = 1.0 * exp(-dot(u.x() - u0, u.x() - u0) / (2.0 * pulseWidth));
056
057    // Output the initial field:
058    std::cout << "Time = 0:\n";
059    std::cout << u << std::endl;
060
061    const Vector<Dim> v(0.2);   // Propagation velocity
062    const double dt = 0.1;      // Timestep
063
064    // Prime the leapfrog by setting the field at the previous timestep using the
065    // initial conditions:
066    uPrev = u;
067
068    // Do a preliminary timestep using forward Euler, using the canned POOMA
069    // stencil-based divergence operator div() for the spatial difference:
070    u -= div<Cell>(v * dt * u);
071
072    // Now use staggered leapfrog (second-order) for the remaining timesteps
073    // The spatial derivative is just the second-order finite difference in the
074    // canned POOMA stencil-based divergence operator div():
075    for (int timestep = 2; timestep <= 1000; timestep++)
076    {
077      uTemp = u;
078      u = uPrev - 2.0 * div<Cell>(v * dt * u);
079      if ((timestep % 100) == 0)
080      {
081        // Output the field at the current timestep:
082        std::cout << "Time = " << timestep*dt << ":\n";
083        std::cout << u << std::endl;
084      }
085      uPrev = uTemp;
086    }
087
088    Pooma::finalize();
089    return 0;
090  }
```

The key lines are 13-15, which define the dimensionality of the simulation, and the size of the domain on which the simulation will be performed. Lines 18-21 then initialize an array of vertex domain objects, the number of elements in which is defined in terms of the `Dim` constant. Similarly, lines 24-32 create a mesh, and a geometry, in a dimension-independent way. Note that when a single value is passed to the constructor of an *N*-dimensional `Vector`, that value is assigned to all of the vector's elements. Note also the use of the vector dot product `dot(Vector<>,Vector<>)` in line 55 to compute the distance from the pulse-center point.

The rest of this program continues in this vein---periodic boundary conditions are set on line 46, for example, and the initial pulse is created on lines 51-55. The result is a program which is only six lines longer than its one-dimensional equivalent, but capable of changing dimension with ease.

Summary

One of the principal motivations behind POOMA is to provide C++ classes which directly address numerical science problems using the language of numerical scientists. The `Field` classes described in this tutorial exemplify this. By managing boundary conditions, and supporting efficient evaluation of differential operators, these classes provide the functionality that modern numerical algorithms require, and allow numerical scientists to concentrate on what they want to calculate, rather than on how it is to be calculated.

[Prev] [Home] [Next]

POOMA Tutorial 8
More on Meshes, Centerings, Geometries, and Fields

Contents:

- Introduction
- Div, Grad, and Averaging
- More on Meshes
- Views and the Loss of Geometry Information
- Operations and Their Results
- Field Stencils
- More on Boundary Conditions
  - Using Pre-Built Boundary Conditions
  - Setting Boundary Conditions on Components
  - Boundary Condition Initialization Functors
  - Writing Boundary Conditions
  - Associating Boundary Conditions with Operators
- Summary

Introduction

The previous tutorial introduced the basic features of POOMA's `Field` classes, and the supporting mesh and geometry classes. This tutorial describes some of the more advanced features of these classes, including centering, differential operators, views, and stencils.

Div, Grad, and Averaging

One way to implement discrete spatial differencing operators is to write data-parallel expressions using indexing objects and offsets, as is shown in the first example of the previous tutorial. In the same way that POOMA provides the `Stencil` class system for `Array`, it provides the `FieldStencil` class for `Field`. This provides an alternative, and more efficient, way to implement spatial differencing operators.

Note: this is an experimental feature in POOMA 2.1 which currently does not work correctly with the parallel or the serial asynchronous schedulers (`configure` options `--parallel --sched async` or `--parallel --sched serialAsync`). Serial code should work for all engine types. These limitations will be addressed in a future version of POOMA.

`FieldStencil` is different from `Stencil` primarily in that it allows the output `Field` to have a different geometry than the input `Field`. Typically, this is useful for implementing operators that go from one centering to another on a mesh.

POOMA provides a small set of canned differential operators that implement various gradient and divergence operators. These are global template functions taking a `ConstField` as input, and returning a `ConstField` with a (possibly) different centering on the same mesh as output. Because they are implemented using `FieldStencil`s, however, these functions do not create temporary objects. Rather, they operate on neighborhoods of values in the input `Field` and return a computed value from each neighborhood. The index location of the output point in the output `Field` is embedded in the

`FieldStencil::operator()` implementation; these `FieldStencil` functors for the POOMA divergence and gradient implementations are parameterized on input and output Centering types with partial specializations. As a result, when these `FieldStencil`-based differential operators are used in expressions with other `Field`s and `Array`s, their operations will all be inlined via the expression-template system.

The interface for the divergence and gradient operators is a pair of global template functions called `div()` and `grad()`. The former takes as its input a `ConstField` of `Vector`s (or `Tensor`s) on a discrete geometry with one centering and returns a `ConstField` of scalars (or `Vector`s). The geometry of the result is the same as that of the input, except possibly for a different centering. The definition of `div()` is as shown below; all of the real work is done in the partial specializations of `Div`'s `operator()`:

```
template<class OutputCentering, class Geometry, class T, class EngineTag>
typename
  View1<FieldStencil<Div<OutputCentering, Geometry, T> >,
    ConstField<Geometry, T, EngineTag> >::Type_t
div(const ConstField<Geometry, T, EngineTag> &f)
{
  typedef FieldStencil<Div<OutputCentering, Geometry, T> > Functor_t;
  typedef ConstField<Geometry, T, EngineTag> Expression_t;
  typedef View1<Functor_t, Expression_t> Ret_t;
  return Ret_t::make(Functor_t(), f);
}
```

The `grad()` function works in a similar way, and has a similar definition. `grad()` takes as its input a `ConstField` of scalars (or `Vector`s) on a discrete geometry with one centering, and returns a `ConstField` of `Vectors` (or `Tensor`s) on a geometry that is the same except (possibly) for the centering. As with `div()`, the real work happens in the partial specializations of `Grad::operator()`:

```
template<class OutputCentering, class Geometry, class T, class EngineTag>
typename
  View1<FieldStencil<Grad<OutputCentering, Geometry, T> >,
    ConstField<Geometry, T, EngineTag> >::Type_t
grad(const ConstField<Geometry, T, EngineTag> &f)
{
  typedef FieldStencil<Grad<OutputCentering, Geometry, T> > Functor_t;
  typedef ConstField<Geometry, T, EngineTag> Expression_t;
  typedef View1<Functor_t, Expression_t> Ret_t;
  return Ret_t::make(Functor_t(), f);
}
```

The underlying `Grad` and `Div` functors' `operator()` methods implement second-order centered finite-difference approximations to the appropriate differential operators. For example, the one-dimensional specialization for `Div` taking a vertex-centered `Field<Vector>` as input, and returning a cell-centered scalar `Field<double>` is:

```
template<class F>
inline OutputElement_t
operator()(const F &f, int i1) const
{
  return
    dot(f(i1    ), Dvc_m[0]/f.geometry().mesh().vertexDeltas()(i1))
  + dot(f(i1 + 1), Dvc_m[1]/f.geometry().mesh().vertexDeltas()(i1));
}
```

Once the syntax is stripped away, this is equivalent to the difference between the values at the vertices *i* and *i+1* (i.e. the left and right neighbors of cell *i*, divided by the vertex-to-vertex spacing. The `Dvc_m` factors are geometrical constants that depend only on the dimensionality.

The following code takes the gradient of a vertex-centered scalar field and produces a cell-centered `Field<Vector>`:

```
Field<DiscreteGeometry<Vert, Mesh_t>, double>             fScalarVert(geomv);
```

```
Field<DiscreteGeometry<Cell, Mesh_t>, Vector<Dim> > fVectorCell(geomc);
fVectorCell = grad<Cell>(fScalarVert)
```

The table below shows the set of input and output `Field` element types, and input and output centerings (on `UniformRectilinearMesh` and `RectilinearMesh`), for which these functors are defined with partial specializations. This set duplicates all the functions provided in version 1 of POOMA. More input and output centering combinations will be added as this version is developed, in particular face, edge, and component-wise centerings such as `VectorFace`.

	Input	Output
Gradient	Scalar/Vert	Vector/Cell
	Scalar/Cell	Vector/Vert
	Scalar/Vert	Vector/Vert
	Scalar/Cell	Vector/Cell
	Vector/Vert	Tensor/Cell
	Vector/Cell	Tensor/Vert
Divergence	Vector/Vert	Scalar/Cell
	Vector/Cell	Scalar/Vert
	Vector/Cell	Scalar/Cell
	Vector/Vert	Scalar/Vert
	Tensor/Vert	Vector/Cell
	Tensor/Cell	Vector/Vert

A related function that POOMA provides is the `average()` function. This function is implemented like, and has an interface similar to, `div()` and `grad()`, but all it calculates is an (optionally weighted) average of `Field` values from one centering to another. The global template function definition for unweighted average is:

```
template<class OutputCentering, class Geometry, class T, class EngineTag>
typename
  View1<FieldStencil<Average<OutputCentering, Geometry, T,
    MeshTraits<typename Geometry::Mesh_t>::isLogicallyRectilinear> >,
    ConstField<Geometry, T, EngineTag> >::Type_t
average(const ConstField<Geometry, T, EngineTag> &f)
{
  typedef FieldStencil<Average<OutputCentering, Geometry, T,
    MeshTraits<typename Geometry::Mesh_t>::isLogicallyRectilinear> >
    Functor_t;
  typedef ConstField<Geometry, T, EngineTag> Expression_t;
  typedef View1<Functor_t, Expression_t> Ret_t;
  return Ret_t::make(Functor_t(), f);
}
```

while that for weighted average is:

```
template<class OutputCentering, class Geometry, class T, class EngineTag,
  class TW, class EngineTagW>
typename
  View2<WeightedFieldStencil<WeightedAverage<OutputCentering, Geometry, T,
    TW, MeshTraits<typename Geometry::Mesh_t>::isLogicallyRectilinear> >,
    ConstField<Geometry, T, EngineTag>,
    ConstField<Geometry, TW, EngineTagW> >::Type_t
average(const ConstField<Geometry, T, EngineTag> &f,
        const ConstField<Geometry, TW, EngineTagW> &weight)
{
  typedef WeightedFieldStencil<WeightedAverage<OutputCentering, Geometry, T,
    TW, MeshTraits<typename Geometry::Mesh_t>::isLogicallyRectilinear> >
    Functor_t;
```

```
  typedef ConstField<Geometry, T, EngineTag> Expression1_t;
  typedef ConstField<Geometry, TW, EngineTagW> Expression1_t;
  typedef View2<Functor_t, Expression1_t, Expression2_t> Ret_t;
  return Ret_t::make(Functor_t(), f, weight);
}
```

The second definition takes an extra argument `weight`, which has the same geometry as the input `Field f`, and multiplies the set of values of the `f` that are combined to produce an output value. The sum of these weighted values are normalized by dividing by the sum of the weight values.

More on Meshes

POOMA's `UniformRectilinearMesh` and `RectilinearMesh` also expose some data arrays that provide such things as cell volumes, surface normal vectors for cell faces, and so on. (These arrays are based on compute engines for the sake of storage efficiency.) There are also methods such as `cellContaining()`, which returns the cell containing a specified point---this is useful in contexts such as particle-mesh interactions. The following table lists the most useful of these; for an up-to-date description the full set, please see the class header files.

Exported typedefs	
`AxisType_t`	The type used to represent the range of a coordinate axis (the mesh class's `T` parameter).
`CellVolumesArray_t`	The type of `ConstArray` returned by `cellVolumes()`.
`CoordinateSystem_t`	The same type as the template parameter `CoordinateSystem`.
`Domain_t`	The type of the mesh's domain. This is currently `Interval<Dim>`.
`PointType_t`	The type of a point (coordinate vector) in the mesh.
`PositionsArray_t`	The type of `ConstArray` returned by `vertexPositions()`.
`SpacingsArray_t`	The type of `ConstArray` returned by `vertexDeltas()`.
`SurfaceNormalsArray_t`	The type of `ConstArray` returned by `cellSurfaceNormals()`.
`SurfaceNormalsArray_t`	The type of `ConstArray` returned by `cellSurfaceNormals()`.
`This_t`	The type of this class.
Exported Enumerations and Constants	
`dimension`	The dimensionality of the mesh (see the note below).
`coordinateDimension`	The dimensionality of the mesh's coordinate system.
Accessors	
`coordinateSystem()`	Returns the mesh's coordinate system.
`origin()`	Returns the mesh origin.
Domain Functions	
`physicalDomain()`	Returns the mesh's domain, excluding its guard layers. This is an indexing object spanning the mesh's vertices, and has type `Domain_t`.
`totalDomain()`	Like `domain()`, but including the mesh's guard layers.
`physicalCellDomain()`	Returns the domain of the mesh's cells.
`totalCellDomain()`	Like `cellDomain()`, but including the mesh's guard layers
Spacing Functions	
`meshSpacing()`	(Defined for `UniformRectilinearMesh` only.) Returns the constant mesh spacings as a coordinate vector of type `PointType_t`.
`vertexDeltas()`	Returns a `ConstArray` of inter-vertex spacings.
Position Functions	
`vertexPositions()`	Returns a `ConstArray` of vertex positions.
Volume Functions	
`cellVolumes()`	Returns a `ConstArray` of cell volumes.
`totalVolumeOfCells()`	Returns the total volume of (a subset of) the mesh.
Cell Surface Functions	
`cellSurfaceNormals()`	Returns a `ConstArray` of surface normals for the cells.

Point Locator Functions	
`cellContaining()`	Returns the indices of the cell containing the specified point, as a `Loc<Dim>`.
`nearestVertex()`	Returns the indices of the vertex nearest the specified point, as a `Loc<Dim>`.
`vertexBelow()`	Returns the indices of the vertex below the specified point, as a `Loc<Dim>`.

Note that the `dimensions` value exported from these logically-rectilinear mesh classes is the `Dim` template parameter for their `Array` data members, such as the array of vertex-vertex mesh spacings returned by `vertexDeltas()`. This value is also the number of integers require to index a single mesh element. While the mesh class's dimension and its spatial dimensionality are the same for logically-rectilinear meshes, an unstructured mesh might well use one-dimensional `Array`s to store data such as vertex positions, despite having a spatial dimensionality of three.

It is always a good idea to use the `typedef`s exported by various classes when declaring objects which will be filled by return values from those objects' accessor functions, or which serve as input for to them. For example, the input argument to `RectilinearMesh::cellContaining()` is `RectilinearMesh::PointType_t`, so the best way to declare variables serving as its input argument is using the exported `typedef` `PointType_t`:

```
const int Dim = 3;
// ...unshown code to set up vertexDomain object...
RectilinearMesh<Dim> mesh(vertexDomain);
RectilinearMesh<Dim>::PointType_t point;
// ...unshown code to set values in the coordinate vector point...
Loc<Dim> whereItsAt = mesh.cellContaining(point);
```

Views and the Loss of Geometry Information

`Field` and `ConstField` support the same sort of view operations as the corresponding array classes: `operator()(Interval)`, `read(Range)`, and `operator()(Interval,int,Range)` all behave as one would expect. However, the result of a view operation on a field is not an array, but rather a new field.

By taking a view of a field, an application is saying that it wants to read or write part of the `Field`'s domain. The physical and total domains of the view are both an `Interval`. The view copies the boundary conditions from the original field. Whether these boundary conditions are applied or not depends on whether the view's base domain---that is, the view's domain mapped back to the index space of the original field---touches the destination domain of one of the boundary conditions.

To make this a bit more concrete, suppose that `f` is an instance of `Field<G,T,E>` for some types `G`, `T`, and E, that `cf` is a `ConstField<G,T,E>`, and that `D` is a domain whose points fit inside the total domain of `f` and `cf`. Then:

- `f(D)` is a `Field<G',T,E'>` representing the view of `f` on `D`;
- `f.read(D)` is a `ConstField<G',T,E'>` representing a read-only view of `f` on `D`;
- `cf(D)` is a `ConstField<G',T,E'>` representing a read-only view of `cf` on `D`;
- `cf.read(D)` is a `ConstField<G',T,E'>` representing a read-only view of `cf` on `D`;
- `f.read()` is a `ConstField<G',T,E'>` representing the view of `f` on the physical domain `PD`;
- `f()` is a `Field<G',T,E'>` representing the view of `f` on the physical domain `PD`;
- `f.readAll()` is a `ConstField<G',T,E'>` representing a read-only view of the field's total domain;
- `f.all()` is a `Field<G',T,E'>` representing a view of the field's total domain;
- `cf()` is a `ConstField<G',T,E'>` representing a read-only view of `f` on the physical domain `PD`;
- `f.array()` is an `Array<dim<PD>,T,E'>` representing an array view of `f` on the physical domain `PD`;
- `f.arrayAll()` is an `Array<dim<TD>,T,E'>` representing an array view of `f` on the total domain `TD`;
- `f.arrayRead()` is a `ConstArray<dim<PD>,T,E'>` representing a read-only array view of `f` on the physical domain `PD`; and
- `f.arrayReadAll()` is a `ConstArray<dim<TD>,T,E'>` representing a read-only array view of `f` on the total domain `TD`.

The exact type of the geometry `G'` resulting from a view of a `Field` depends on the original geometry `G` and the domain type `D`. In POOMA 2.1, if `G` is a `DiscreteGeometry<Centering,Mesh>` and `D` is an `Interval`, `G'` will be a `DiscreteGeometry<Centering,MeshView<Mesh>>` (i.e. a fully-functional discrete geometry with the same

centering and a view of the part of the mesh described by the `Interval`). This works because all meshes in POOMA 2.1 are logically rectilinear. Therefore, it is possible to deduce the connectivity of part of a mesh.

However, if `D` is a more complicated domain, such as a `Range` or indirection list, there is no sensible way to deduce connectivity automatically, and so the notions of a mesh and centering are lost. POOMA represents this notion by introducing a "no geometry" `Geometry` class. For all non-`Interval`-based views, `G'` evaluates to a `NoGeometry<N>`, where `N` is the dimensionality.

Another complicated case is a binary operation involving two `Field`s. If the two `Field`s do not have the same geometry, there is no way to know what the geometry of the resulting `Field` should be. (The library could make an arbitrary choice, such as always using the geometry from the left operand, but this would be wrong as often as it was right). If the two `Field`s have the same geometry type, it is still not possible to know until run-time whether they really hold equivalent geometry objects. Lacking a clear idea of how to construct the geometry, the library again opts for the straightforward solution of returning a `NoGeometry<N>` geometry. Note, however, that if only one of the operands is a Field, the library can know unambiguously what geometry to use. Therefore, these operations preserve geometry information.

Given the complications associated with deducing the `Geometry`, one could ask why not just make the view of a `Field` an `Array`? The reason is the automatic boundary condition updates discussed in the previous tutorial. If a `Field` was also an `Array`, applications would not be able to update boundary conditions through views. It therefore makes sense that views, along with all the other field-related entities that can find themselves at the leaf of a PETE expression tree, be `Field`s of some sort. Also, as a general rule, POOMA attempts to preserve as much information as possible when applying views.

Operations and Their Results

The rules governing the results of operations on `Field`s are more complex than those for `Array`s because `Field`s incorporate geometries. As with `Array`s, all operations involving at least one `Field` result in a `Field`. However, it is not always possible to preserve geometry information. The table below illustrates this, using the following declarations (where all objects are 2-dimensional unless otherwise noted):

```
Field<Geometry_t,Vector<2> >f
Field<Geometry_t>            g
Interval<2>                  I
Interval<1>                  J
Range<2>                     R
Array<2>                     a
```

It may be useful to compare this table to the one given in the second tutorial.

Operation	Example	Output Type
Taking a view of the field's physical domain	`f()`	`Field<ViewGeometry_t,Vector<2>,BrickView<2,true>>`
Taking a view of the field's total domain	`f.all()`	`Field<ViewGeometry_t,Vector<2>,BrickView<2,true>>`
Taking a view using an `Interval`	`f(I)`	`Field<ViewGeometry_t,Vector<2>,BrickView<2,true>>`
Taking a view using a `Range`	`f(R)`	`Field<NoGeometry<2>,Vector<2>,BrickView<2,false>>`
Taking a slice	`f(2,J)`	`Field<NoGeometry<1>,Vector<2>,BrickView<2,true>>`
Indexing	`f(2,3)`	`Vector<2>&`
Taking a read-only view of the field's physical domain	`f.read()`	`ConstField<ViewGeometry_t,Vector<2>,BrickView<2,true>>`
Taking a read-only view of the field's total domain	`f.readAll()`	`ConstField<ViewGeometry_t,Vector<2>,BrickView<2,true>>`

Taking a read-only view using an `Interval`	`f.read(I)`	`ConstField<ViewGeometry_t,Vector<2>,BrickView<2,true>>`
Taking a read-only view using a `Range`	`f.read(R)`	`ConstField<NoGeometry<2>,Vector<2>,BrickView<2,false>>`
Taking a read-only slice	`f.read(2,J)`	`ConstField<NoGeometry<1>,Vector<2>,BrickView<2,true>>`
Reading an element	`f.read(2,3)`	`Vector<2>`
Taking a component view	`f.comp(1)`	`Field<Geometry_t,double, CompFwd<Engine<2,Vector<2>,Brick>,1>>`
Taking a read-only component view	`f.compRead(1)`	`ConstField<Geometry_t,double, CompFwd<Engine<2,Vector<2>,Brick>,1>>`
Applying a unary operator or function	`sin(f)`	`ConstField<Geometry_t,Vector<2>, ExpressionTag<UnaryNode<FnSin, ConstField<Geometry_t,Vector<2>,Brick>>>>`
Applying a binary operator or function involving two `Field`s	`f + g`	`ConstField<NoGeometry<2>,Vector<2>,ExpressionTag< BinaryNode<OpAdd, ConstField<Geometry_t,Vector<2>,Brick>, ConstField<Geometry_t,double,Brick>>>>`
Applying a binary operator or function to a `Field` and a scalar	`2 * f`	`ConstField<Geometry_t,Vector<2>,ExpressionTag< BinaryNode<OpMultiply, Scalar<int>, ConstField<Geometry_t,double,Brick>>>>`
Applying a binary operator or function to a `Field` and an `Array`	`a + f`	`ConstField<Geometry_t,Vector<2>,ExpressionTag< BinaryNode<OpAdd, ConstArray<2,double,Brick>>, ConstField<Geometry_t,double,Brick>>>>`
Note: If `Geometry_t` is a `DiscreteGeometry<C,M>`, where `M` is a logically rectilinear mesh, then `ViewGeometry_t` will be a `DiscreteGeometry<C,MeshView<M>>`.		

As before, indexing produces an element type while all other operations yield a `Field` or `ConstField` with a different engine, perhaps a different element type, and perhaps a new geometry. `ConstField`s result when the operation is read-only in nature. Notice that some of the operations return a `Field` with a geometry of type `NoGeometry<N>`, where `N` is dimensionality. The reason for this, and the difficulties that can ensue, were discussed earlier.

Field Stencils

The tutorial on pointwise functions introduced the `Stencil` class that is used to implement point-by-point calculations on `Array`s. A closely related class called `FieldStencil` serves the same purpose for `Field`s. Its basic interface and implementation are similar to that of `Stencil`, but it has special capabilities to handle `Field`'s geometric properties. These in turn imply some extra requirements on the interface of user-defined functors for `FieldStencil`.

`FieldStencil` class is parameterized the same way as `Stencil`:

```
template<class Functor>
struct FieldStencil
{
  ...
}
```

Any functor class that is to serve as the template parameter to `FieldStencil` must have certain characteristics; in particular, it must define an appropriate set of `operator()` methods. In order to see what these are, consider the definition of the divergence stencil functor `Div`:

```
template<class OutputCentering, class Geometry, class T>
```

```
class Div {};
```

The definition of the partial specialization in question is given in `src/Field/DiffOps/Div.URM.h`, and is:

```
template<int Dim, class T1, class T2, class EngineTag>
class Div<Cell,
  DiscreteGeometry<Vert, RectilinearMesh<Dim, Cartesian<Dim>, T1 > >,
  Vector<Dim, T2> >
{
public:
  typedef Cell OutputCentering_t;
  typedef T2 OutputElement_t;

  // Constructors.
  Div()
  {
    T2 coef = 1.0;
    for (int d = 1; d < Dim; d++)
    {
      coef *= 0.5;
    }
    for (int d = 0; d < Dim; d++)
    {
      for (int b = 0; b < (1 << Dim); b++)
      {
        int s = ( b & (1 << d) ) ? 1 : -1;
        Dvc_m[b](d) = s*coef;
      }
    }
  }

  // Extents
  int lowerExtent(int d) const { return 0; }
  int upperExtent(int d) const { return 1; }

  // One dimension
  template<class F>
  inline OutputElement_t
  operator()(const F &f, int i1) const
  {
    return (dot(f(i1    ), Dvc_m[0]/f.geometry().mesh().vertexDeltas()(i1)) +
            dot(f(i1 + 1), Dvc_m[1]/f.geometry().mesh().vertexDeltas()(i1)));
  }

  // Two dimensions
  template<class F>
  inline OutputElement_t
  operator()(const F &f, int i1, int i2) const
  {
    const typename F::Geometry_t::Mesh_t::SpacingsArray_t &vD =
      f.geometry().mesh().vertexDeltas();
    return (dot(f(i1    , i2    ), Dvc_m[0]/vD(i1, i2)) +
            dot(f(i1 + 1, i2    ), Dvc_m[1]/vD(i1, i2)) +
            dot(f(i1    , i2 + 1), Dvc_m[2]/vD(i1, i2)) +
            dot(f(i1 + 1, i2 + 1), Dvc_m[3]/vD(i1, i2)));
  }

  // Three dimensions
  template<class F>
```

```
  inline OutputElement_t
  operator()(const F &f, int i1, int i2, int i3) const
  {
    const typename F::Geometry_t::Mesh_t::SpacingsArray_t &vD =
      f.geometry().mesh().vertexDeltas();
    return (dot(f(i1    , i2    , i3    ), Dvc_m[0]/vD(i1, i2, i3)) +
            dot(f(i1 + 1, i2    , i3    ), Dvc_m[1]/vD(i1, i2, i3)) +
            dot(f(i1    , i2 + 1, i3    ), Dvc_m[2]/vD(i1, i2, i3)) +
            dot(f(i1 + 1, i2 + 1, i3    ), Dvc_m[3]/vD(i1, i2, i3)) +
            dot(f(i1    , i2    , i3 + 1), Dvc_m[4]/vD(i1, i2, i3)) +
            dot(f(i1 + 1, i2    , i3 + 1), Dvc_m[5]/vD(i1, i2, i3)) +
            dot(f(i1    , i2 + 1, i3 + 1), Dvc_m[6]/vD(i1, i2, i3)) +
            dot(f(i1 + 1, i2 + 1, i3 + 1), Dvc_m[7]/vD(i1, i2, i3)));
  }

private:
  // Geometrical constants for derivatives:
  Vector<Dim,T2> Dvc_m[1<<Dim];
};
```

The `operator()` method is defined for 1, 2, and 3 integer indices. These make this functor general enough to handle all types of input `Field`s (whose types are instances of the member template's `F` parameter), as long as the `Field` type's individual elements can be indexed by 1, 2, or 3 integers. The exported typedef `InputField_t`, however, restricts this particular `Div` functor to input `Field`s using the POOMA `DiscreteGeometry<Vert,RectilinearMesh>` geometry type.

The implementations of `operator()` assume that the elemental type of the input `Field` is a `Vector`, for which the dot product of an element with the `Dvc_m` member `Vector` (componentwise-divided by the local vertex-vertex mesh spacing value) makes sense. The `Dvc_m` data member is time-independent state data useful for this particular divergence stencil implementation.

The required methods `lowerExtent()` and `upperExtent()` are very much like their `Stencil` counterparts. Because the output `Field` type of the `FieldStencil` has a different centering than the input `Field` type, however, care must be taken when interpreting these stencil widths. In this example, the input centering is `Vert` and the output centering is `Cell`. The value of `lowerExtent(d)` and `upperExtent(d)` are therefore 0 and 1 respectively, even though this is a centered-difference stencil, for which you might expect the lower extent to be -1 rather than zero.

To understand the values of `lowerExtent(d)` and `upperExtent(d)` for this cell-to-vertex stencil example, consider Figure 1, which is appropriate for any single value of the argument `d`.

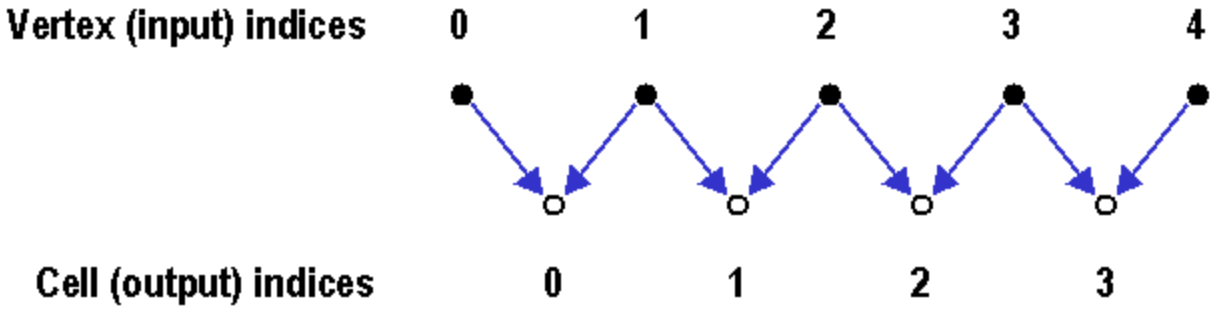


Figure 1: `lowerExtent()` and `upperExtent()` are asymmetrical in value for this `Div` stencil example even though it is a centered difference formula, because of the centering effects on the index spaces. (The values are 0 and 1 respectively.) The blue arrows show the pairs of input-centering-index-space indices which combine to produce a value with a single output-centering-index-space index. The differencing is centered (combine the values from two vertices centered about each cell center), but the index-space offsets in the input index space are asymmetrical because of the different domain sizes.

The value returned by `lowerExtent(d)` is then the maximum positive integer offset from the element indexed by integer `i` in the input `Field`'s index space along dimension `d` used in outputting the element indexed by integer `i` in the output `Field`'s index space along dimension `d`. The (physical) domains of the input and output `Field`s along each dimension are of different

lengths (because there is one more vertex than cell center along a dimension), so it is important to think carefully about what this implies about the stencil-width methods and the implementation of the `operator()` methods.

Applications can construct `FieldStencil` functors that are parameterized on functors such as the `Div` functor above, then invoke them via `FieldStencil::operator()` in the same way as was done with the `Stencil<LaplaceStencil>` functor in the `Array Stencil` example:

```
// Create the geometries, assuming RectilinearMesh object mesh:
typedef RectilinearMesh<Dim, Cartesian<Dim> > Mesh_t
DiscreteGeometry<Vert, Mesh_t> geomv(mesh, GuardLayers<Dim>(1));
DiscreteGeometry<Cell, Mesh_t> geomc(mesh, GuardLayers<Dim>(1));

// Make the Fields (default EngineTag type):
Field<DiscreteGeometry<Vert, Mesh_t>, Vector<Dim> > vv(geomv);
Field<DiscreteGeometry<Cell, Mesh_t>, double > sc(geomc);

// Make the divergence FieldStencil object, using the Div class defined above:
typedef Div<Cell, DiscreteGeometry<Vert, Mesh_t>, Vector<Dim> > Div_t;
FieldStencil<Div_t> divVV2SC();

// Divergence, Vector/Vert-->Scalar/Cell
sc = divV2SC(fv);
```

Programmers may also find it convenient to create wrappers by defining global template functions which internally construct appropriate `FieldStencil<class Stencil>` objects, like the `div()` function described above.

More on Boundary Conditions

Whenever POOMA encounters a data-parallel expression involving fields, boundary conditions may be applied. However, POOMA tries to ensure that these calculations are only done when absolutely necessary. Before evaluating an expression, POOMA asks each of the boundary conditions for each of the fields on the right-hand side of an assignment operator whether the source domain has been modified since the last time the boundary condition has been evaluated, and whether the domain for the data parallel expression touches the destination domain. The boundary condition is re-computed only if both of these are true. Otherwise, evaluation proceeds directly to the data-parallel expression.

Delaying evaluation in this way can forestall a lot of unnecessary calculation. The price for this is that programmers must be careful when writing scalar code, because scalar expression evaluation does not automatically trigger the update of field boundary conditions. To force calculation of all of a field's boundary conditions explicitly, an application must call the method `Field::applyBoundaryConditions()`. In particular:

- when reading from values in the destination domains of the boundary conditions, call `applyBoundaryConditions()` before the scalar loop; and
- when writing to values in the source domains, call `applyBoundaryConditions()` after the scalar loop.

In addition, boundary conditions are not automatically evaluated before a field is printed. Applications should therefore call `applyBoundaryConditions()` before output statements to ensure that the boundary values displayed are up to date..

Using Pre-Built Boundary Conditions

POOMA includes a number of pre-built boundary conditions for use with fields and the supplied rectilinear meshes. For example, the following code sets the guard layers of a `Dim`-dimensional field `f` to zero:

```
for (int d = 0; d < 2 * Dim; d++)
{
  f.addBoundaryCondition(ZeroFaceBC(d));
}
```

All of the pre-built boundary conditions apply themselves to a particular face of the rectilinear computational domain. For each component direction, there is a high and a low face. For a `Dim`-dimensional field, faces are numbered consecutively from `0` to `2*Dim-1`. The faces for each axis are numbered consecutively, with the low face having the lower (even) number. Thus, the

coordinate direction and whether the face is the high or low face is calculated as follows:

```
int direction = face / 2;
bool isHigh   = (face & 1);
```

The high face in the Y direction therefore has a face index of 3 (second axis, second face).

The pre-built boundary conditions supported by POOMA are:

- `ConstantFaceBC<T>(int face, T constant, bool enforceConstantBoundary = false);`
- `LinearExtrapolateFaceBC(int face);`
- `NegReflectFaceBC(int face, bool enforceZeroBoundary = false);`
- `PeriodicFaceBC(int face);`
- `PosReflectFaceBC(int face, bool enforceZeroBoundary = false);`
- `ZeroFaceBC(int face, bool enforceZeroBoundary = false);`

`ConstantFaceBC<T>` represents a Dirichlet boundary condition on a domain (i.e. one which keeps the value on that face constant). The constructor switch `enforceConstantBoundary` allows the boundary condition to enforce that the mesh-boundary value is constant, i.e. to determine whether the boundary condition writes into the guard layers, or into the actual physical domain. This affects only vertex-centered field values/components because the boundary is defined to be the last vertex. The `T` template parameter is the type of the constant value.

`LinearExtrapolateFaceBC` takes the values of the last two physical elements, and linearly extrapolates from the line through them out to all the guard elements. This is independent of centering. Like the other boundary conditions in this release of POOMA, it applies only to logically rectilinear domains.

`NegReflectFaceBC` represents an antisymmetric boundary condition on a logically rectilinear domain where the value on that face is assumed to be zero. As with the `ConstantFaceBC` boundary condition, the constructor switch `enforceZeroBoundary` allows the boundary condition to enforce that the boundary value is zero. This affects only vertex-centered field values/components because the boundary is defined to be the last vertex.

`PeriodicFaceBC` represents a periodic boundary condition in one direction of a logically rectilinear domain.

`PosReflectFaceBC` represents a symmetric boundary condition on a logically rectilinear domain; the face itself may take on any value. The constructor switch `enforceZeroBoundary` allows the boundary condition to enforce that the boundary value is zero. This affects only vertex-centered field values/components because the boundary is defined to be the last vertex.

`ZeroFaceBC` represents a zero Dirichlet boundary condition on a logically rectilinear domain. The constructor switch `enforceZeroBoundary` allows the boundary condition to enforce that the mesh-boundary value is zero. This affects only vertex-centered field values/components because the boundary is defined to be the last vertex.

Setting Boundary Conditions on Components

Applications often need to apply different boundary conditions to different components of a `Vector` or `Tensor` field. In POOMA, this is accomplished using the `ComponentBC` adaptor, which works by taking a component view of the field and then applying the specified boundary condition to that view. Consider the example:

```
// Create the geometry.
typedef RectilinearCentering<D, VectorFaceRCTag<D> > Centering_t;
DiscreteGeometry<Centering_t, UniformRectilinearMesh<D> >
  geom(mesh, GuardLayers<D>(1));

// Make the field.
Field<DiscreteGeometry<Centering_t, UniformRectilinearMesh<D> >, Vector<D> >
  f(geom);

// Add componentwise boundary conditions.
typedef ComponentBC<1,NegReflectFaceBC> NegReflectFace_t;
typedef ComponentBC<1,PosReflectFaceBC> PosReflectFace_t;
for (int face = 0; face < 2 * D; face++)
```

```
{
  int direction = face / 2;
  for (int c = 0; c < D; c++)
  {
    if (c == direction)
      f.addBoundaryCondition(NegReflectFace_t(c, face));
    else
      f.addBoundaryCondition(PosReflectFace_t(c, face));
    }
}
```

This adds $2D^2$ boundary conditions for each of the D components at the high and low faces in each of the D coordinate directions. The `ComponentBC` class is templated on the number of indices (1 for `Vector`s and 2 for `Tensor`s) and the boundary condition category (e.g., `PosReflectFaceBC`). The constructor arguments are the 1 or 2 indices specifying the components followed by the constructor arguments for the boundary condition.

Boundary Condition Initialization Functors

It is often easiest for an application to set all of a field's boundary conditions at once. POOMA supports this by allowing boundary conditions to be initialized using a functor, as in:

```
f.addBoundaryConditions(AllZeroFaceBC());
```

This sets zero boundary conditions for all faces and components of the field `f` in a single statement. (Note the 's' at the end of the method name `addBoundaryConditions()`). The definition of the functor `AllZeroFaceBC` is simply:

```
class AllZeroFaceBC
{
public:
  AllZeroFaceBC(bool enforceZeroBoundary = false)
  : ezb_m(enforceZeroBoundary) { }

  template<class Geometry, class T, class EngineTag>
  void operator()(Field<Geometry, T, EngineTag> &f) const
  {
    for (int i = 0; i < 2 * Geometry::dimensions; i++)
    {
      f.addBoundaryCondition(ZeroFaceBC(i, ezb_m));
    }
  }
private:
  bool ezb_m;
};
```

Constructor arguments for the individual boundary conditions are specified when constructing the functor. The actual boundary conditions are added in the functor's `operator()` method, which is called internally by the field.

This release of POOMA predefines the functors listed below. Their effects can be inferred by comparing them with the the boundary conditions given in the previous table.

- `AllConstantFaceBC<T>(T constant, bool enforceConstantBoundary = false);`
- `AllLinearExtrapolateFaceBC();`
- `AllNegReflectFaceBC(bool enforceZeroBoundary = false);`
- `AllPeriodicFaceBC();`
- `AllPosReflectFaceBC(bool enforceZeroBoundary = false);`
- `AllZeroFaceBC(bool enforceZeroBoundary = false);`

Writing Boundary Conditions

In order to add a new type of boundary condition for POOMA, an application must define two classes: a boundary condition category, and the boundary condition itself. The boundary condition category class is the user interface for the boundary condition, and is simply a lightweight functor. (Classes like `ConstantFaceBC<T>` are boundary condition category classes of this kind.) For example, a boundary condition category for the following spatially-dependent two-dimensional boundary condition:

```
f(face) = 100 * x(face) * y(face)
```

could be written as:

```
class PositionFaceBC : public BCondCategory<PositionFaceBC>
{
public:
  PositionFaceBC(int face)
  : face_m(face)
  {}

  int face() const
  {
    return face_m;
  }

private:
  int face_m;
};
```

Notice that the class inherits from a version of `BCondCategory` templated on itself, but is otherwise quite straightforward.

The actual boundary condition is a specialization of the `BCond` class, which has the general template definition:

```
template<class Subject, class Category>
class BCond;
```

The `Subject` is the class of field that the boundary condition is to be applied to. POOMA needs to know this type exactly because it must be able to apply the boundary condition using PETE's data-parallel machinery.

To continue with the previous example, a specialization for the spatially-dependent boundary condition that is appropriate for two-dimensional multi-patch fields is:

```
typedef Field<
  DiscreteGeometry<Vert, UniformRectilinearMesh<2> >,
  double, MultiPatch<UniformTag, Brick> > FieldType_t;

template<>
class BCond<FieldType_t, PositionFaceBC> :
  public FieldBCondBase<FieldType_t>
{
public:
  // Constructor computes the destination domain
  BCond(const FieldType_t &f, const PositionFaceBC &bc)
  : FieldBCondBase<FieldType_t>(f, f.totalDomain())
  {
    int d = bc.face() / 2;
    int hiFace = bc.face() & 1;
    int layer;
    if (hiFace)
    {
```

```
        layer = destDomain()[d].last();
      }
      else
      {
        layer = destDomain()[d].first();
      }
      destDomain()[d] = Interval<1>(layer, layer);
    }

    void applyBoundaryCondition()
    {
      subject()(destDomain()) = 100.0 * subject().x(destDomain()).comp(0) *
        subject().x(destDomain()).comp(1);
    }

    BCond<FieldType_t, PositionFaceBC> *retarget(const FieldType_t &f) const
    {
      return new BCond<FieldType_t, PositionFaceBC>(f, bc_m);
    }
  };
```

This could obviously be written more generally, but is sufficient to illustrate the concepts. Notice that this is a full specialization of the `BCond` template. Such specializations must inherit from the base class `FieldBCondBase`, which is templated on the field type.

The constructor for `FieldBCondBase` takes up to three arguments: the field, the initial value of the destination domain, and the initial value of the source domain. The last two domain arguments are optional. If they are not specified, the domains are initialized to be empty. The field argument can be subsequently accessed using the `subject()` member, the destination domain can be accessed using the `destDomain()` method, and the source domain can be accessed using the `srcDomain()` method.

The destination domain is the domain that fully bounds the region where the boundary condition is setting values. The source domain bounds the region where the boundary condition gets values to compute with. In this example, the destination domain is the single guard layer outside the physical domain for the specified face. There is no source domain because the destination values are not computed using other values. This is not the case with, for instance, the `PeriodicFaceBC` boundary condition, where periodicity is enforced by copying values from one place to another.

In many cases, the source and destination domains exactly define where values are read from, and where they are written. However, it is important to realize that POOMA treats these as bounding boxes. This means that for fields based on rectilinear meshes, the types of these domains will be `Interval<Geometry::dimensions>`. If a boundary condition doesn't write or read from domains specified by an `Interval` (e.g., a `Range`), this domain must be computed and stored specially.

For example, suppose an application had a boundary condition that set every other point in the guard layers. The destination domain member `destDomain()` would still return an `Interval`, since it represents a bounding box, not the actual domain. These two entities are the same for all of the boundary conditions that this release of POOMA contains; however, future versions may relax this constraint.

In addition to a constructor, a boundary value class must have a method called `applyBoundaryCondition()`, which must contain the code that actually evaluates the boundary condition, and a method called `retarget()`, which makes a new boundary condition using a different subject and the internal data of the current object. The example above uses straightforward data-parallel to syntax apply the boundary conditions. More sophisticated examples are included in the `src/BConds` directory in the release.

Associating Boundary Conditions with Operators

By default, POOMA associates boundary conditions with fields. This was done to allow automatic computation of boundary conditions and for compatibility with POOMA R1. An alternative approach is associating boundary conditions with operators. The source code below, taken from `examples/Field/Laplace2`, illustrates how this is done:

```
001  #include "Pooma/Fields.h"
002  #include "Utilities/Clock.h"
```

```
003
004  #include <iostream>
005
006  // Convenience typedefs.
007
008  typedef ConstField<
009    DiscreteGeometry<Vert, UniformRectilinearMesh<2> > > ConstFieldType_t;
010
011  typedef Field<
012    DiscreteGeometry<Vert, UniformRectilinearMesh<2> > > FieldType_t;
013
014  // The boundary condition.
015
016  class PositionFaceBC : public BCondCategory<PositionFaceBC>
017  {
018  public:
019
020    PositionFaceBC(int face) : face_m(face) { }
021
022    int face() const { return face_m; }
023
024  private:
025
026    int face_m;
027  };
028
029  template<>
030  class BCond<FieldType_t, PositionFaceBC>
031    : public FieldBCondBase<FieldType_t>
032  {
033  public:
034
035    BCond(const FieldType_t &f, const PositionFaceBC &bc)
036    : FieldBCondBase<FieldType_t>
037        (f, f.totalDomain()), bc_m(bc) { }
038
039    void applyBoundaryCondition()
040    {
041      int d = bc_m.face() / 2;
042      int hilo = bc_m.face() & 1;
043      int layer;
044      Interval<2> domain(subject().totalDomain());
045      if (hilo)
046        layer = domain[d].last();
047      else
048        layer = domain[d].first();
049
050      domain[d] = Interval<1>(layer, layer);
051      subject()(domain) = 100.0 * subject().x(domain).comp(0) *
052        subject().x(domain).comp(1);
053    }
054
055    BCond<FieldType_t, PositionFaceBC> *retarget(const FieldType_t &f) const
056    {
057      return new BCond<FieldType_t, PositionFaceBC>(f, bc_m);
058    }
059
060  private:
061
```

```
062   PositionFaceBC bc_m;
063 };
064
065 // The stencil.
066
067 class Laplacian
068 {
069 public:
070
071   typedef Vert OutputCentering_t;
072   typedef double OutputElement_t;
073
074   int lowerExtent(int) const { return 1; }
075   int upperExtent(int) const { return 1; }
076
077   template<class F>
078   inline OutputElement_t
079   operator()(const F &f, int i1, int i2) const
080   {
081     return 0.25 * (f(i1 + 1, i2) + f(i1 - 1, i2) +
082       f(i1, i2 + 1) + f(i1, i2 - 1));
083   }
084
085   template<class F>
086   static void applyBoundaryConditions(const F &f)
087   {
088     for (int i = 0; i < 4; i++)
089       {
090         BCondItem *bc = PositionFaceBC(i).create(f);
091         bc->applyBoundaryCondition();
092         delete bc;
093       }
094   }
095 };
096
097 void applyLaplacian(const FieldType_t &l, const FieldType_t &f)
098 {
099   Laplacian::applyBoundaryConditions(f);
100   l = FieldStencil<Laplacian>()(f);
101 }
102
103 int main(
104     int argc,
105     char *argv[]
106 ){
107     // Set up the library
108     Pooma::initialize(argc,argv);
109
110     // Create the physical domains:
111
112     // Set the dimensionality:
113     const int nVerts = 100;
114     Loc<2> center(nVerts / 2, nVerts / 2);
115     Interval<2> vertexDomain(nVerts, nVerts);
116
117     // Create the (uniform, logically rectilinear) mesh.
118     Vector<2> origin(1.0 / (nVerts + 1)), spacings(1.0 / (nVerts + 1));
119     typedef UniformRectilinearMesh<2> Mesh_t;
120     Mesh_t mesh(vertexDomain, origin, spacings);
```

```
121
122       // Create a geometry object with 1 guard layer to account for
123       // stencil width:
124       typedef DiscreteGeometry<Vert, UniformRectilinearMesh<2> > Geometry_t;
125       Geometry_t geom(mesh, GuardLayers<2>(1));
126
127       // Create the Fields:
128
129       // The voltage v(x,y) and a temporary vTemp(x,y):
130       FieldType_t v(geom), vTemp(geom);
131
132       // Start timing:
133       Pooma::Clock clock;
134       double start = clock.value();
135
136       // Load initial condition v(x,y) = 0:
137       v = 0.0;
138
139       // Perform the Jacobi iteration. We apply the Jacobi formula twice
140       // each loop:
141       double error = 1000;
142       int iteration = 0;
143       while (error > 1e-6)
144         {
145           iteration++;
146
147           applyLaplacian(vTemp, v);
148           applyLaplacian(v, vTemp);
149
150           // The analytic solution is v(x, y) = 100 * x * y so we can test the
151           // error:
152
153           // Make sure calculations are done prior to scalar calculations.
154           Pooma::blockAndEvaluate();
155
156           const double solution = v(center);
157           const double analytic = 100.0 * v.x(center)(0) * v.x(center)(1);
158           error = abs(solution - analytic);
159           if (iteration % 1000 == 0)
160             std::cout << "Iteration: " << iteration << "; "
161                 << "Error: " << error << std::endl;
162         }
163
164       std::cout << "Wall clock time: " << clock.value() - start << std::endl;
165       std::cout << "Iteration: " << iteration << "; "
166                 << "Error: " << error << std::endl;
167
168       Pooma::finalize();
169       return 0;
170   }
```

This is a simple Jacobi saolver for Laplace's equation using the `PositionFaceBC` boundary condition discussed above. Lines 110-130 set up the mesh, the geometry, and the `Brick`-engine-based `Field`. Notice that we do not add any boundary conditions to this field, but we do reserve one layer of external guard layers (line 125) We then initialize the `Field` `v` and begin iterating. Since we need a temporary field to store the result of the Laplacian stencil, we can efficiently perform two applications of the stencil for each loop (lines 147 and 148). We know that the analytic solution of this problem is *v(x, y) = 100 x y*, so we can monitor and report the error in lines 156-161. Finally, we use the POOMA `Clock` class to monitor wall-clock time (lines 132-134 and 164).

The function `applyLaplacian` (lines 97-101) takes a `Field` to assign to and a `Field` to stencil as arguments. This is where the boundary conditions are applied, follwed by the stencil. The `Field` stencil object `Laplace` is straightforward, except for the static function `applyBoundaryConditions` that, on the fly, creates boundary conditions for each face of the input field `f` and applies them.

Future versions of POOMA will better support this paradigm of associating boundary conditions with operators.

Summary

Fields are among the most important structures in physics, and POOMA's `Field` classes are one of the things that make it more than just another array package. While the facilities introduced in this tutorial and the preceding one are more complex than some other parts of POOMA, all of their complexity is necessary in order to give applications programmers both expressive power and high performance.

[Prev] [Home] [Next]

POOMA Tutorial 9
Particles

Contents:

Introduction
Overview
Attributes
Layout
Derivation
Synchronization and Related Issues
Example: Simple Harmonic Oscillator
Boundary Conditions
Example: Elastic Collision
Summary

Introduction

Particles are primarily used in one of two ways in large scientific applications. The first is to track sample particles using Monte Carlo techniques, for example, to gather statistics that describe the conditions of a complex physical system. Particles of this kind are often referred to as *tracers*. The second is to perform direct numerical simulation of systems that contain discrete point-like entities such as ions or molecules.

In both scenarios, the application contains one or more sets of particles. Each set has some data associated with it that describes its members' characteristics, such as mass and charge. Particles typically exist in a spatial domain, and they may interact directly with one another or with field quantities defined on that domain.

This tutorial gives an overview of POOMA's support for particles, then discusses some implementation details. The classes introduced in this tutorial are illustrated by two short programs: one that tracks particles under the influence of a simple one-dimensional harmonic oscillator potential, and another that models particles bouncing off the walls of a closed three-dimensional box. The next tutorial then shows how particles and fields can be combined to create complete simulation applications.

Overview

POOMA's `Particles` class is a container for a heterogeneous collection of particle attributes. The class uses dynamic storage for particle data (in the form of `DynamicArray`s), so that particles can be added or deleted as necessary. It contains a layout object that manages the distribution of particle data across multiple patches, and it applies boundary conditions to particles when attribute data values exceed a prescribed range. In addition, global functions are provided for interpolating data between particle and field element positions.

Each `Particles` object keeps a list of pointers to its elements' attributes. When an application wants to add or delete a particle, it invokes a method on the `Particles` object, which delegates the call to the layout object for the contained attributes. `Particles` also provides a member function called `sync()`, which the application invokes in order to update the particle count and data distribution across contexts and to apply the boundary conditions.

Applications can define a new type of particles collection by deriving from the `Particles` class. The derived class declares data members for the attributes needed to characterize this type of particle; the types of these data members are discussed below. The constructor for this class calls `Particles::addAttribute` to register each attribute and add it to the list. In this way, the `Particles` class can be extended by the application to accommodate any sort of particle description.

The distribution of particle data stored in `DynamicArray`s is directed by a particle layout class. (The details of the mechanism used to specify layout and other information for `Particles` classes are discussed below.) Each particle layout class employs a particular strategy to determine the patch in which a particle's data should be stored. For instance, `SpatialLayout` keeps each particle in the patch that contains field data for elements that are nearest to the particle's current spatial position. This strategy is useful for cases where the particles need to interact with field data or with particles nearby to them.

Attributes

Each particle attribute is implemented as a `DynamicArray`, a class derived from the one-dimensional specialization of POOMA's `Array` class. `DynamicArray` extends the notion of a one-dimensional array to allow applications to add or delete elements at will. When particles are destroyed, the empty slots left behind can be filled by moving elements from the end of the list (backfill) or by sliding all the remaining elements over and preserving the existing order (shift up). At the same time, `DynamicArray`s can be used in data-parallel expressions in the same way as ordinary `Array`s, so that the application can update particle attributes such as position and velocity using either a single statement or a loop over individual particles.

At first glance, it might seem more sensible to have applications define a type `T` that stores all the attribute data for one particle in a single data structure, and then use this as a template argument to the `Particles` class, which would store a `DynamicArray` of values of this type. POOMA's designers considered this option, but discarded it. The reason is that most compute-intensive operations in scientific applications are implemented as loops in which one or more separate attributes are read or written. In order to make the evaluation of expressions involving attributes as efficient as possible, it is therefore important to ensure that data are arranged as separate one-dimensional arrays for each attribute, rather than as a single array of structures with one structure per particle. This arrangement makes common cases such as:

```
for (int i=0; i<n; ++i)
{
  x[i] += dt * vx[i];
  y[i] += dt * vy[i];
}
```

run more quickly, as it makes much better use of the cache.

Layout

As mentioned above, each `Particles` object uses a layout object to determine in which patch a particle's data should be stored. The layout manages the program's requests to re-arrange particle data. With `SpatialLayout`, for example, the application provides a particle position attribute which is used to determine how particle data should be distributed. The particle layout then directs the `Particles` object to move particle data from one patch to another as dictated by its strategy. The `Particles` object in turn delegates this task to the layout object for the particle attributes, which tells each of the attributes using this layout to move their data as needed. All of this is handled by a single call to `Particles::sync()`, which in turn calls `Particles::swap()` to actually move particle data around.

Derivation

In general, creating a new `Particles` class is a three-step process. The first step is to declare a traits class whose `typedef`s specify the type of engine the particle attributes are to use and the way the data for those attributes is to be distributed. An example of such a traits class is the following:

```
struct MyParticleTraits
{
  typedef MultiPatch<GridTag,Brick> AttributeEngineTag_t;
  typedef UniformLayout             ParticleLayout_t;
};
```

This traits class will be used to specialize the `Particles` class template when an application class representing a concrete set of particles is derived from it. `Particles` uses public `typedef`s to give sensible names to these traits parameters, so that the derived application-level class can access them (as shown below). For the application developer's convenience, a set of pre-defined particle traits classes with specific choices of attribute engine and particle layout type are provided in the header file `src/Particles/CommonParticleTraits.h`. These define combinations of shared brick and multi-patch brick engines with both uniform and spatial layouts, and include the following:

Name	AttributeEngineTag_t	ParticleLayout_t
SharedBrickUniform	SharedBrick	UniformLayout
SharedBrickSpatial <Cent,Mesh,FieldLayout>	SharedBrick	SpatialLayout< DiscreteGeometry<Cent,Mesh>, FieldLayout>
MPBrickUniform	MultiPatch<GridTag,Brick>	UniformLayout
MPBrickSpatial <Cent,Mesh,FieldLayout>	MultiPatch<GridTag,Brick>	SpatialLayout< DiscreteGeometry<Cent,Mesh>, FieldLayout>

The `SharedBrick` engine type is just like `Brick`, except that the engine's layout can be shared by other engines constructed with the same layout argument. The effect of this is that the layout of all of the attributes remains synchronized. `SharedBrick` should only be used when running serially; otherwise, applications should use `MultiPatch`.

The second step is to derive a class from `Particles`. The new class can be templated on whatever the developer desires, as long as a traits class type is provided for the template parameter of the `Particles` base class. In the example below, the new class being derived from `Particles` is templated on the same traits class as `Particles`. For the sake of convenience, `typedef`s may be provided for the instantiated parent class and for its layout type. The constructor for the application class then usually takes a concrete layout object of the type specified in the `typedef` above as a constructor argument:

```
template <class PT>
class MyParticles : public Particles<PT>
{
public:
  // instantiated type of parent class
  typedef Particles<PT> Base_t;

  // type of layout (from traits class via parent class)
  typedef typename Base_t::ParticleLayout_t ParticleLayout_t;

  // type of attribute engine tag (from traits class via parent class)
  typedef typename Base_t::AttributeEngineTag_t EngineTag_t;

  // some particle attributes as public data members
  DynamicArray<double, EngineTag_t> charge;
  DynamicArray<double, EngineTag_t> mass;
  DynamicArray<int, EngineTag_t>    count;

  // constructor invokes Particles(layout) to cache layout
  MyParticles(const ParticleLayout_t &layout)
  : Particles<PT>(layout)
  {
    // register attributes
    addAttribute(charge);
    addAttribute(mass);
    addAttribute(count);
  }
};
```

Note that the attribute elements in this example have different element types, i.e., `charge` and `mass` are `double`, while `count` is `int`. Attribute elements may in general have any type, including any user-defined type.

Finally, the application class `MyParticles` is instantiated with the traits class `MyParticleTraits` to create an actual set of particles. An actual layout is declared first, and it is passed as a constructor argument to the instance of the application-level class to control the distribution of particle data between patches. This layout object typically has one or more constructor arguments that specify such things as the number of patches the particles are to be distributed over:

```
int main()
```

```
{
  const int numPatches = 10;
  MyParticleTraits::ParticleLayout_t layout(numPatches);
  MyParticles<MyParticleTraits>      particles(layout);
}
```

While this may seem complex at first, each level of indirection or generalization is needed in order to provide flexibility. The type of engine and layout to be used, for example, could be passed directly as template parameters to `Particles`, rather than being combined together in a traits class. However, this would make user-level code fragile in the face of future changes to the library: if other traits are needed later, they can be added to the traits class in one place, rather than needing to be specified every time something is derived from `Particles`. This bundling also makes it easier to specify the same basic properties (engine and layout) for two or more interacting `Particles`-derived classes.

Synchronization and Related Issues

For efficiency reasons, `Particles` does not automatically move particle data between patches after every operation, but instead waits for the application to call the method `sync()`. `Particles` can also be configured to cache requests to delete particles, rather than deleting them immediately.

`Particles::sync()` is a member template, i.e., it is templated on its single argument. This argument must be one of the particle set's attributes. `SpatialLayout` assumes that the attribute given to `sync()` is the particles' positions, and uses it to update the distribution of particle data so that particles are located on the same patch as nearby field data. Applications must therefore be careful not to mistakenly pass a non-spatial attribute, such as temperature or pressure, to `SpatialLayout`.

`UniformLayout`, which divides particles as evenly as possible between patches, without regard for spatial position, only uses the attribute passed to `sync()` as a template for the current distribution of particle data. Any attribute with the same distribution as the actual particle data can therefore be used.

The use of a parameter in `Particles::sync()` is one important difference between the implementation of particles in this version of POOMA and its predecessor. In the old design, all `Particles` classes came with a pre-defined attribute `R` that was the particles' position, which synchronization always referred to. The new scheme allows applications to switch the attribute that is used to represent the position, e.g., to switch back and forth between a "current" position attribute `currpos` and a "new" position attribute `newpos`. It also allows particles to be weighted according to some attribute, so that the distribution scheme load-balances by weight.

Of course, before particle data can be (re-)distributed, the particles themselves must be created. `Particles` provides two methods for doing this. The first, `globalCreate(num,renum)`, creates a specified number of particles, spread as evenly as possible across all patches. The particles are normally renumbered after the creation operation, although this can be overridden by passing `false` as a second parameter to the method.

`Particles::create(num, patch, renum)`, on the other hand, creates a specified number of particles within the local context, and adds them to either the last local patch (if the `patch` argument is negative) or to a specific patch (if `patch` is non-negative). The particles are renumbered after this operation unless `false` is passed as a third parameter to this method.

After particles have been created (or destroyed), they must be renumbered to ensure that each has a unique ID. In general, the `renumber()` method surveys all the patches to find out what the current local domain of each patch is. It then reconstructs a global domain across all the patches, effectively renumbering the particles from *0* to *N-1*, where *N* is the total number of particles. The more complex `sync()` method applies the particle boundary conditions, performs any deferred particle destroy requests, swaps particles between patches according to the particle layout strategy, and then renumbers the particles by calling `renumber()`. Programs should therefore call `renumber()` if they have only created or destroyed particles, but have not done deferred destroy requests, modified particle attributes in a way that would require applying boundary conditions (or have no boundary conditions), and do not need to swap particles.

If a program does not (implicitly or explicitly) call `renumber()` after creating or destroying particles, the global domain for the particles will be incorrect. If the program then tries to read or write a view of a particle attribute by indexing with some domain object, it will not get the right section of the data. This failure could be silent if the view that the program requests exists. Alternatively, the requested view could be outside of the global domain (because `renumber()` was not called to update the global domain), in which case the layout object for the particle attribute will suffer a run-time assertion failure.

There are also two ways to destroy particles. The first way, which always destroys the particles immediately, is implemented by the method `Particles::destroy(domain, patchId, renum)`. If the `patchId` parameter is negative (which is the default), the `domain` is assumed to specify a global numbering of particles. If `patchId` is non-negative, then `domain` is assumed to be a local numbering for that patch, i.e., one in which the first particle in the patch has index 0.

Since this method modifies the `Particles` object right away, the default behavior of this method is to renumber particles after it has finished destroying the specified particles. This can be overridden by passing `false` as the last parameter to the call.

The second particle destruction method is `Particles::deferredDestroy(domain, patch)`. This is new in this release, and only does deferred destruction, i.e., only caches the requested indices for use later when `performDestroy()` is called. (Since this method doesn't actually destroy particles right away, there is no need for it to call `renumber()`. The `performDestroy()` method, which causes the cached destruction requests to be executed, always performs renumbering.)

As noted above, `Particles::globalCreate()` normally calls `renumber()` to update the global domain of the particle attributes after the particles have been created, but before the program tries to do computations involving their attributes. The reason for this is that while `globalCreate()` allocates space for the new particle data and updates the local domain of the patch or patches on which creation was done, the global domain across all the patches of data is not updated until the call to `renumber()`. If the global domain is not up to date, the program cannot correctly access the i^{th} particle's data or evaluate a data-parallel expression.

Example: Simple Harmonic Oscillator

The example for this tutorial simulates the motion of particles under the influence of a simple one-dimensional harmonic oscillator potential. The code, which is included in the release in the `examples/Particles/Oscillation` directory, is as follows:

```
001  #include <iostream>
002  #include <stdlib.h>
003
004  #include "Pooma/Particles.h"
005  #include "Pooma/DynamicArrays.h"
006
007  // Dimensionality of this problem
008  static const int PDim = 1;
009
010  // A traits class specifying the engine and layout of a Particles class.
011  template <class EngineTag>
012  struct PTraits
013  {
014    // The type of engine to use in the particle attributes.
015    typedef EngineTag AttributeEngineTag_t;
016
017    // The type of particle layout to use.  Here we use a UniformLayout,
018    // which divides the particle data up so as to have an equal number
019    // on each patch.
020    typedef UniformLayout ParticleLayout_t;
021  };
022
023  // A Particles subclass that defines position and velocity as
024  // attributes.
025  template <class PT>
026  class Quanta : public Particles<PT>
027  {
028  public:
029    // Useful things to extract from the base class
030    typedef Particles<PT>                          Base_t;
031    typedef double                                 AxisType_t;
032    typedef typename Base_t::ParticleLayout_t      ParticleLayout_t;
033    typedef typename Base_t::AttributeEngineTag_t AttributeEngineTag_t;
034    enum { dimensions = PDim };
035
036    // Constructor sets up layouts and registers attributes
037    Quanta(const ParticleLayout_t &pl)
038    : Particles<PT>(pl)
039    {
040      addAttribute(x);
```

```
041     addAttribute(v);
042   }
043
044   // X position and velocity are attributes (would normally be
045   // private, with accessor methods)
046   DynamicArray< Vector<dimensions, AxisType_t>, AttributeEngineTag_t > x;
047   DynamicArray< Vector<dimensions, AxisType_t>, AttributeEngineTag_t > v;
048 };
049
050 // Engine tag type for attributes.  Here we use a MultiPatch engine
051 // with the patches being Bricks of data, and a GridTag, which allows
052 // the patches to possibly be of differing sizes.  This is important
053 // since we may not have the same number of particles in each patch.
054 typedef MultiPatch<GridTag, Brick> AttrEngineTag_t;
055
056 // The particle traits class and layout type for this application
057 typedef PTraits<AttrEngineTag_t> PTraits_t;
058 typedef PTraits_t::ParticleLayout_t PLayout_t;
059
060 // Simulation control constants
061 const double omega      = 2.0;
062 const double dt         = 1.0 / (50.0 * omega);
063 const int nParticle     = 100;
064 const int nPatch        = 4;
065 const int nIter         = 500;
066
067 // Main simulation routine.
068 int main(int argc, char *argv[])
069 {
070   // Initialize POOMA and Inform object for output to terminal
071   Pooma::initialize(argc, argv);
072   Inform out(argv[0]);
073   out << "Begin Oscillation example code" << std::endl;
074
075   // Create a uniform layout object to control particle positions.
076   PLayout_t layout(nPatch);
077
078   // Create Particles, using our special subclass and the layout
079   typedef Quanta<PTraits_t> Particles_t;
080   Particles_t p(layout);
081
082   // Create particles on one patch, then re-distribute (just to show off)
083   p.create(nParticle, 0);
084   for (int ip=0; ip<nPatch; ++ip)
085   {
086     out << "Current size of patch " << ip << " = "
087         << p.attributeLayout().patchDomain(ip).size()
088         << std::endl;
089   }
090
091   out << "Resyncing particles object ... " << std::endl;
092   p.sync(p.x);
093
094   // Show re-balanced distribution.
095   for (int ip=0; ip<nPatch; ++ip)
096   {
097     out << "Current size of patch " << ip << " = "
098         << p.attributeLayout().patchDomain(ip).size()
099         << std::endl;
100   }
101
```

```
102    // Randomize positions in domain [-1,+1], and set velocities to zero.
103    // This is done with a loop because POOMA does not yet have RNGs.
104    typedef Particles_t::AxisType_t Coordinate_t;
105    Vector<PDim, Coordinate_t> initPos;
106    srand(12345U);
107    for (int ip=0; ip<nParticle; ++ip)
108    {
109      for (int idim=0; idim<PDim; ++idim)
110      {
111        initPos(idim) = 2.0*(rand() / static_cast<Coordinate_t>(RAND_MAX))-1.0;
112      }
113      p.x(ip) = initPos;
114      p.v(ip) = Vector<PDim, Coordinate_t>(0.0);
115    }
116
117    // print initial state
118    out << "Time = 0.0:" << std::endl;
119    out << "Quanta positions:" << std::endl << p.x << std::endl;
120    out << "Quanta velocities:" << std::endl << p.v << std::endl;
121
122    // Advance particles in each time step according to:
123    //         dx/dt = v
124    //         dv/dt = -omega^2 * x
125    for (int it=0; it<numit; ++it)
126    {
127      p.x = p.x + dt * p.v;
128      p.v = p.v - dt * omega * omega * p.x;
129      out << "Time = " << (it+1)*dt << ":" << std::endl;
130      out << "Quanta positions:" << std::endl << p.x << std::endl;
131      out << "Quanta velocities:" << std::endl << p.v << std::endl;
132    }
133
134    // Finalize POOMA
135    Pooma::finalize();
136    return 0;
137  }
```

As discussed earlier, the program begins by creating a traits class that `typedef`s the names `AttributeEngineTag_t` and `ParticleLayout_t` (lines 11-21). An application-specific class called `Quanta` is then derived from `Particles`, without specifying the traits to be used (lines 25-48). This class declares two attributes, to store the particles' *x* coordinate and velocity. The body of its constructor (lines 40-41) adds these attributes to its attribute list, while passing the actual layout object specified by the application up to `Particles`.

Lines 54, 57 and 58 create some convenience `typedef`s for the engine and layout that the application will use. Lines 61-65 then define constants describing both the physical parameters to the problem (such as the oscillation frequency) and the computational parameters (the number of particles, the number of patches, etc.). In a real application, many of these values would be variables, rather than hard-wired constants.

After the POOMA library is initialized (line 71), an `Inform` object is created to manage output. (See the appendix on I/O for a description of this class.) An actual layout is then created (line 76), and used to create an actual set of particles (line 80). The particles themselves are created by the call to `Particles::create()` on line 83. The output on lines 84-89 shows that all particles are initially created in the zeroth patch.

The `sync()` call on line 92 redistributes particles across the available patches according to their *x* coordinates. As the output from lines 95-100 shows, this load-balances the particles as evenly as possible.

The particle positions are randomized on lines 107-115. (A loop is used here because random number generation has not yet been integrated into the expression evaluation machinery in this release of POOMA.) After some more output to show the particles' initial positions, the application finally enters the main timestep loop (lines 125-132). In each time step, particle positions and velocities are updated under the influence of a simple harmonic oscillator force, and then printed out. Once the specified number of timesteps has been executed, the library is shut down (line 135) and the application exits.

Boundary Conditions

In addition to an `AttributeList`, each `Particles` object also stores a `ParticleBCList` of boundary conditions to be applied to the attributes. These are generalized boundary conditions in the sense that they can be applied not only to a particle position attribute, but to any sort of attribute or expression involving attributes. POOMA provides typical particle boundary conditions including periodicity, reflection, absorption, reversal (reflection of one attribute and negation of another), and kill (destroying a particle). Boundary conditions can be updated explicitly by calling `Particles::applyBoundaryConditions()`, or implicitly by calling `Particles::sync()` (which performs the same operations, along with several others).

Each boundary condition is assembled by first constructing an instance of the type of boundary condition desired, then invoking the `addBoundaryCondition()` member function of `Particles` with three parameters: the subject of the boundary condition (i.e., the attribute or expression to be checked against the range), its object (the attribute to be modified when the subject is outside the range), and the actual boundary condition object. The boundary condition is then applied each time the `sync()` function is invoked.

The subject and object of a boundary condition are usually the same, but this is not required. In one common case, the subject is an expression involving particle attributes, while the object is the `Particles` object itself. For example, an application's boundary condition might specify that particles are to be deleted if their kinetic energy goes above some limit. The subject would be the expression `0.5*m*v*v`, and the object could be either one of the particle attributes (because deleting a particle from one attribute automatically deletes it from all the others) or the `Particles` object itself. The object cannot be the expression `0.5*m*v*v` because that is a `ConstArray` and cannot be modified.

Another case involves the reversal boundary condition, which is used to make particles bounce off walls. Bouncing not only reflects the particle position back inside the wall, but also reverses the particle's velocity component in that direction. The reversal boundary condition therefore needs an additional object besides the original subject.

POOMA provides the pre-defined boundary condition classes listed in the table below.

Class	**Behavior**
`AbsorbBC<T>(T min, T max)`	Keeps attributes within given limits `min` or `max`. If they cross the given boundaries, their values are changed to the given limiting value.
`KillBC<T>(T min, T max)`	If particles cross outside the given boundary, they are destroyed by putting their index in the deferred destroy list.
`PeriodicBC<T>(T min, T max)`	Keeps attributes within a given periodic domain.
`ReflectBC<T>(T min, T max)`	Reflects an attribute back if it crosses outside of the given boundary.
`ReverseBC<T>(T min, T max)`	Reverses (negates) the value of the object attribute if it crosses outside the given domain, and reflects the value of the subject attribute.

Example: Elastic Collision

As an example of how particle boundary conditions are used, consider a set of particles bouncing around in a box in three dimensions. `examples/Particles/Bounce/Bounce.cpp` shows how this can be implemented using POOMA for the case of perfectly elastic collisions. The code is:

```
001  #include "Pooma/Particles.h"
002  #include "Pooma/DynamicArrays.h"
003  #include "Tiny/Vector.h"
004  #include "Utilities/Inform.h"
005  #include <iostream>
006  #include <stdlib.h>
007
008
009  // Dimensionality of this problem
010  static const int PDim = 3;
011
012  // Particles subclass with position and velocity
013  template <class PT>
014  class Balls : public Particles<PT>
```

```
015  {
016  public:
017    // Typedefs
018    typedef Particles<PT>                            Base_t;
019    typedef typename Base_t::AttributeEngineTag_t  AttributeEngineTag_t;
020    typedef typename Base_t::ParticleLayout_t      ParticleLayout_t;
021    typedef double                                   AxisType_t;
022    typedef Vector<PDim,AxisType_t>                  PointType_t;
023
024    // Constructor: set up layouts, register attributes
025    Balls(const ParticleLayout_t &pl)
026    : Particles<PT>(pl)
027    {
028      addAttribute(pos);
029      addAttribute(vel);
030    }
031
032    // Position and velocity attributes (as public members)
033    DynamicArray< PointType_t, AttributeEngineTag_t >  pos;
034    DynamicArray< PointType_t, AttributeEngineTag_t >  vel;
035  };
036
037  // Use canned traits class from CommonParticleTraits.h
038  // MPBrickUniform provides MultiPatch Brick Engine for
039  // particle attributes and UniformLayout for particle data.
040  typedef MPBrickUniform PTraits_t;
041
042  // Type of particle layout
043  typedef PTraits_t::ParticleLayout_t ParticleLayout_t;
044
045  // Type of actual particles
046  typedef Balls<PTraits_t> Particle_t;
047
048  // Number of particles in simulation
049  const int NumPart = 100;
050
051  // Number of timesteps in simulation
052  const int NumSteps = 100;
053
054  // Number of patches to distribute particles across.
055  // Typically one would use one patch per processor.
056  const int numPatches = 16;
057
058  // Main simulation routine
059  int main(int argc, char *argv[])
060  {
061    // Initialize POOMA and output stream
062    Pooma::initialize(argc, argv);
063    Inform out(argv[0]);
064
065    out << "Begin Bounce example code" << std::endl;
066    out << "-------------------------" << std::endl;
067
068    // Create a particle layout object for our use
069    ParticleLayout_t particleLayout(numPatches);
070
071    // Create the actual Particles object (but not the particles as yet)
072    Particle_t balls(particleLayout);
073
074    // Create some particles, recompute the global domain, and initialize
075    // the attributes randomly.  The globalCreate call will create an equal
```

```
076    // number of particles on each patch.  The particle positions are initialized
077    // within a 12 X 20 X 28 domain, and the velocity components are all
078    // in the range -4 to +4.
079    balls.globalCreate(NumPart);
080    srand(12345U);
081    Particle_t::PointType_t initPos, initVel;
082    for (int i = 0; i < NumPart; ++i)
083    {
084      for (int d = 0; d < PDim; ++d)
085      {
086        initPos(d) = ( (d+1) * 8.0 + 4.0 ) * rand() /
087                     static_cast<Particle_t::AxisType_t>(RAND_MAX);
088        initVel(d) = 8.0 * rand() /
089                     static_cast<Particle_t::AxisType_t>(RAND_MAX) - 4.0;
090      }
091      balls.pos(i) = initPos;
092      balls.vel(i) = initVel;
093    }
094
095    // Display the particle positions and velocities.
096    out << "Timestep 0: " << std::endl;
097    out << "Ball positions: "  << balls.pos << std::endl;
098    out << "Ball velocities: " << balls.vel << std::endl;
099
100    // Set up a reversal boundary condition, so that particles will
101    // bounce off the domain boundaries.
103    Particle_t::PointType_t lower, upper;
104    for (int d = 0; d < PDim; ++d)
105    {
106      lower(d) = 0.0;
107      upper(d) = (d+1) * 8.0 + 4.0;
108    }
109    ReverseBC<Particle_t::PointType_t> bounce(lower, upper);
110    balls.addBoundaryCondition(balls.pos, balls.vel, bounce);
111
112    // Advance simulation stepwise
113    for (int it=1; it <= NumSteps; ++it)
114    {
115      // Advance ball positions (timestep dt = 1)
116      balls.pos += balls.vel;
117
118      // Invoke boundary conditions
119      balls.applyBoundaryConditions();
120
121      // Print out the current particle data
122      out << "Timestep " << it << ": " << std::endl;
123      out << "Ball positions: " << balls.pos << std::endl;
124      out << "Ball velocities: " << balls.vel << std::endl;
125    }
126
127    // Shut down POOMA and exit
128    Pooma::finalize();
129    return 0;
130  }
```

After defining the dimension of the problem (line 10), this program defines a class `Balls`, which represents the set of particles (lines 13-35). Its two attributes represent the particles' positions and velocities (lines 33-34). Note how the type of engine used for evaluating these attributes is defined in terms of the types exported by the traits class with which `Balls` is instantiated (`AttributeEngineTag_t`, line 19), while the type used to represent the points is defined in terms of the dimension of the problem (line 22), rather than being made 1-, 2-, or 3-dimensional explicitly. This style of coding makes it much easier to change the simulation as the program evolves.

Rather than defining a particle traits class explicitly, as the oscillation example above did, this program uses one of the pre-defined traits class given in `src/Particles/CommonParticleTraits.h`. For the purposes of this example, a multipatch brick engine is used for particle attributes, and particle data is laid out uniformly. Once again, a `typedef` is used to create a symbolic name for this combination, so that the program can be updated by making a single change in a single location.

Lines 43-56 then define the types used in the simulation, and the constants that control the simulation's evolution. It would be possible to shorten this part of the program by combining some of these type definitions (as on line 43), but readability would suffer.

The main body of the program follows; as usual, it begins by initializing the POOMA library, and creating an output handler of type `Inform` (lines 62-63). Line 69 then creates a layout object describing the domain of the problem.

The particles object itself comes into being on line 72, although the actual particles aren't created until line 79. Recall that by default, `globalCreate()` (re-)numbers the particles by calling `Particles`' `renumber()` method. As discussed earlier, this could be prevented by passing `false` as a second parameter to `globalCreate()`, i.e., by calling `globalCreate(N, false)`. Lines 80-93 then randomize the balls' initial positions and velocities.

Lines 103-110 are the most novel part of this simulation, as they create reflecting boundary conditions for the simulation, and add them to the `balls` object. Lines 103-108 defines where particles bounce; again, this is done in a dimension-independent fashion in order to make code evolution as easy as possible. Line 104 turns `upper` and `lower` into a reversing boundary condition, which line 105 then adds to `balls`. The main simulation loop now consists of nothing more than advancing the balls in each time step, and calling `sync()` to enforce the boundary conditions.

Summary

Particles are a fundamental construct in physical calculations. POOMA's `Particles` class, and the classes that support it, allow programmers to create and manage sets of particles both efficiently and flexibly. While doing this is a multi-step process, the payoff as programs are extended and updated is considerable. The list below summarizes the most important aspects of `Particles`' interface.

- `Particles<PL>::initialize(PL &layout)`: Initialize the particles object with the given particle layout. This should be used if the `Particles` object was created with the default constructor.
- `size()`: Return the current total number of particles, correct since the last `renumber()`.
- `domain()`: Return the one-dimensional domain of the particle attributes (the `Interval<1>` `0...size()-1`).
- `attributes()`: Return the number of registered attributes.
- `addAttribute(attrib)`: Add the given attribute (should be a `DynamicArray` of the proper engine type) to the `Particles`' attribute list.
- `removeAttribute(attrib)`: Remove the given attribute from the `Particles`' attribute list.
- `sync(posattrib)`: Apply boundary conditions, carry out cached destroys, swap particles, and renumber particles (in that order).
- `swap(posattrib)`: Move particle data between patches as specified by the particle layout strategy (uniform or spatial) and renumber particles.
- `applyBoundaryConditions()`: Apply the boundary conditions to the current attributes, without renumbering or destroying particles.
- `performDestroy()`: Destroy any particles that were specified in previous `deferredDestroy()` requests.
- `renumber()`: Recalculate the per-patch and total domain of the system by inspecting the `Particles`' attribute layout.
- `create(N, patch, renum)`: Create *N* particles in the specified patch (and optionally renumber). If `patch` and `renum` are omitted, this creates particles in the last patch, so as not to disturb the numbering of existing particles.
- `globalCreate(N, renum)`: Create *N/P* particles in the *P* patches that the `Particles` object occupies (and optionally renumber).
- `destroy(domain, patchId, renum)`: Immediately destroy particles in the specified domain, and optionally renumber. The domain may be a one-dimensional range of particle index numbers or a list of index numbers. (See the note below on the `patchId` parameter.)
- `deferredDestroy(domain, patch)`: Put the indices of the particles in the given domain in the deferred destroy list of the `Particles` object, so that they will be destroyed by the next call to `performDestroy()`. (See the note below on the `patchId` parameter.)
- `addBoundaryCondition(Subj, Obj, BCobj)` and `addBoundaryCondition(Subj, BCobj)`: Add a new boundary condition that depends on the subject `Subj` and affects the object `Obj`.

- `removeBoundaryCondition(i)` and `removeBoundaryConditions()`: Delete the i^{th} boundary condition, or all boundary conditions.

Note: if the `patchId` given to `destroy()` or `deferredDestroy()` is negative, the `domain` argument must specify a global domain (i.e., global numbering). If the argument is non-negative, the domain is interpreted as being local, i.e., the index 0 refers to the first particle in that patch.

[Prev] [Home] [Next]

POOMA Tutorial 10
Particles and Fields

Contents:

Introduction
Particle/Field Interpolation
Laying Out Particles and Fields
Example: Particle-in-Cell Simulation
Summary

Introduction

The previous tutorials have described how POOMA represents fields and particles. This tutorial shows how the two can be combined to create complete simulations of complex physical systems. The first section describes how POOMA interpolates values when gathering and scattering field and particle data. This is followed by a look at the in's and out's of layout, and a medium-sized example that illustrates how these ideas fit together.

Particle/Field Interpolation

POOMA's `Particles` class is designed to be used in conjunction with its `Field`s. *Interpolators* are the glue that bind these together, by specifying how to calculate field values at particle (or other) locations that don't happen to lie exactly on mesh points.

Interpolators are used to gather values to specific positions in a field's spatial domain from nearby field elements, or to scatter values from such positions into the field. The interpolation stencil describes how values are translated between field element locations and arbitrary points in space. An example of using this kind of interpolation is particle-in-cell (PIC) simulations, in which charged particles move through a discretized domain. The particle interactions are determined by scattering the particle charge density into a field, solving for the self-consistent electric field, and gathering that field back to the particle positions. The last example in this tutorial describes a simulation of this kind.

POOMA currently offers three types of interpolation stencils: nearest grid point (NGP), cloud-in-cell (CIC), and subtracted dipole scheme (SUDS). NGP is a zeroth-order interpolation that gathers from or scatters to the field element nearest the specified location. CIC is a first-order scheme that performs linear weighting among the 2^D field elements nearest the point in D-dimensional space. SUDS is also first-order, but it uses just the nearest field element and its two neighbors along each dimension, so it is only a 7-point stencil in three dimensions. Other types of interpolation schemes can be added in a straightforward manner.

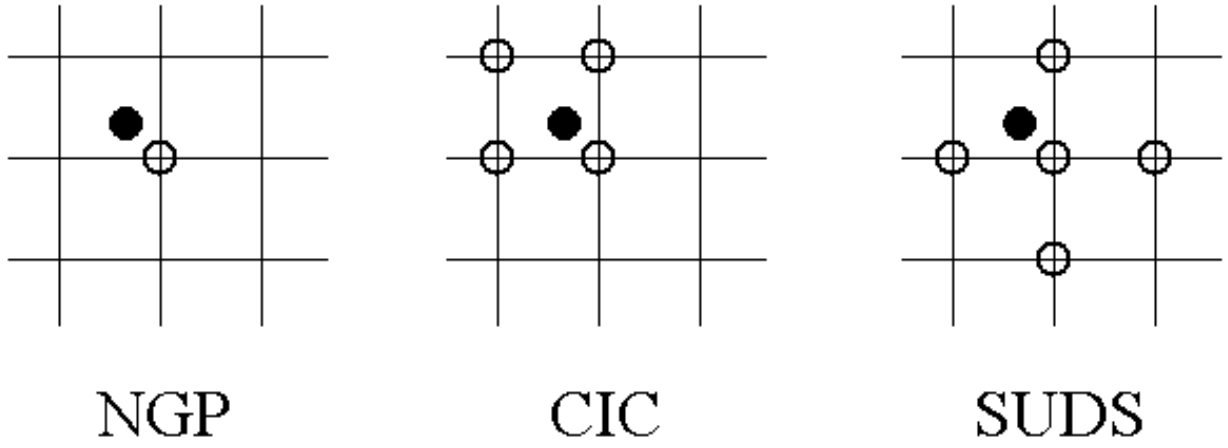


Figure 1: Interpolation strategies. Black dots show particle positions, and open circles are the interpolation stencil points.

Interpolation is invoked by calling the global functions `gather()` and `scatter()`, both of which take four arguments:

1. the particle attribute to be gathered to or scattered from (usually a single `DynamicArray`, although one could scatter an expression involving `DynamicArray`s as well, since the evaluation of this expression just produces a temporary one-dimensional `ConstArray`);
2. the `Field` to be gathered from or scattered to;
3. the particle positions (normally a `DynamicArray` that is a member of a `Particles`-derived class); and
4. an interpolator tag object of type NGP, CIC or SUDS. These tag objects are defined in the header files `InterpolatorNGP.h`, `InterpolatorCIC.h`, and `InterpolatorSUDS.h` respectively.

An example of this is:

```
gather(P.efd, Efield, P.pos, CIC());
```

where `P` is a `Particles` subclass object whose attributes are `efd` for storing the gathered electric field from the `Field` `Efield` and `pos` for the particle positions. The default constructor of the interpolator `CIC` is used to create a temporary instance of the class to pass to `gather()`, telling it which interpolation scheme to use.

The particle attribute and position arguments passed to `gather()` and `scatter()` should have the same layout, and the positions must refer to the geometry of the `Field` being used. The interpolator will compute the required interpolated values for the particles on each patch. These functions assume each particle is only interacting with field elements in the `Field` patch that exactly corresponds to the particle patch. Thus, applications must use the `SpatialLayout` particle layout strategy and make sure that the `Field` has enough guard layers to accommodate the interpolation stencil.

In addition to the basic `gather()` and `scatter()` functions, POOMA offers some variants that optimize other common operations. The first of these, `scatterValue()`, scatters a single value into a `Field` rather than a particle attribute with different values for each particle. Its first argument is a single value with a type that is compatible with the `Field` element type.

The other three optimized methods are `gatherCache()`, `scatterCache()`, and `scatterValueCache()`. Each of these has two overloaded variants, which allow applications to cache and reuse interpolated data, such as the nearest grid point for each particle and the distance from the particle's position to that grid point. The difference between the elements of each overloaded pair of methods is that one takes both a particle position attribute and a particle interpolator cache attribute among its arguments, while the other takes only the cache attribute. When the first of these is called, it caches position information in the provided cache attribute. When the second is called with that cache attribute as an argument, it re-uses that information. This can speed up computation considerably, but it is important to note that applications can only do this safely when the particle positions are guaranteed not to have

changed since the last interpolation.

Laying Out Particles and Fields

The use of particles and fields together in a single application brings up some issues regarding layout that do not arise when either is used on its own. There are two characteristics of `Engine`s that must be considered in order to determine whether they can be used for attributes in `Particles` objects:

> *1. Can the engine use a layout that is "shared" among several engines of the same category, such that the size and layout of the engine is synchronized with the other engines using the layout?* If this is the case, then creation, destruction, repartitioning, and other operations are done for all the shared engines. Particles require all their attributes to use a shared layout, so only engines that use a shared layout can be used for particle attributes. The only engines with this capability in this release of POOMA (i.e., the only engines that are usable in `Particles` attributes) are `SharedBrick` (using a `SharedDomainLayout` layout) and some specializations of `MultiPatch`.
>
> `MultiPatch` can use several different types of layouts and single-block engines, and all `MultiPatch`s use a shared layout. However, only the `MultiPatch<GridTag,*>` types of `MultiPatch` engines are useful for `Particles` attributes, since only that engine type can have patches of varying size. Future releases of POOMA will add other layouts, such as `TileLayout`, that will also be useful for attributes.
>
> Note that `MultiPatch<UniformTag,Brick>` can *not* be used for particle attributes, as it uses a `UniformGridLayout`. While that layout is "shared", `UniformGridLayout` is not useful for particles because it requires all patches to have the same size, and particle attribute patches change their size dynamically.
>
> *2. How many patches can the engine have?* A `SharedBrick` can only have one patch, but a `Grid`-based `MultiPatch` can have several patches. Either one can be used in serial or in parallel, but their efficiency will differ. If individual particle attribute expressions will normally be run in parallel, the application should use a `MultiPatch`. Otherwise, it should use a `SharedBrick`.

Implicit in the discussion above is the fact that there are actually three different types of layout classes that an application programmer must keep in mind:

1. the layout for the particle attributes;
2. the layout for the `Field` given to the particle `SpatialLayout` (which is used to determine the layout of the space in which the particles move around); and
3. the actual `SpatialLayout` that connects the info about the `Field` layout to the `Particles` attribute layout.

The only thing that needs to match between the attribute and `Field` layouts is the number of patches, which must be the same. The engine type (and thus the layout type) of the attributes and of the field do not have to match. An application could therefore use a `SharedBrick` engine for particle attributes, and a `MultiPatch<UniformTag,Brick>` for the `Field`, as long as the `MultiPatch` engine uses just one patch (since `SharedBrick` can only have one patch).

Note once again that in the simple case of a `UniformLayout`, applications do not need to worry about the `Field` layout type, only the particle attributes' layout (which still needs to be shared) and the particle layout (in this case, `UniformLayout`). This commonly arises during the prototyping (i.e., pre-parallel) stages of application development.

Example: Particle-in-Cell Simulation

Our third and final example of this important class is a particle-in-cell program, which simulates the motion of charged particles in a static sinusoidal electrical field in two dimensions. This example brings together the `Field` classes of the preceding tutorials with this tutorial's `Particles` class.

Because this example is longer than the others in these tutorials, it will be described in sections. For a unified listing of the source code, please see the file `examples/Particles/PIC2d/PIC2d.cpp` in the distribution.

The first step is to include all of the usual header files:

```
001  #include "Pooma/Particles.h"
002  #include "Pooma/DynamicArrays.h"
003  #include "Pooma/Fields.h"
004  #include "Utilities/Inform.h"
005  #include <iostream>
006  #include <stdlib.h>
007  #include <math.h>
```

Once this has been done, the application can define a traits class for the `Particles` object it is going to create. As always, this contains `typedef`s for `AttributeEngineTag_t` and `ParticleLayout_t`. The traits class for this example also includes an application-specific `typedef` called `InterpolatorTag_t`, for reasons discussed below.

```
008  template <class EngineTag, class Centering, class MeshType, class FL,
009            class InterpolatorTag>
010  struct PTraits
011  {
012    // The type of engine to use in the attributes
013    typedef EngineTag AttributeEngineTag_t;
014
015    // The type of particle layout to use
016    typedef SpatialLayout<DiscreteGeometry<Centering,MeshType>,FL>
017      ParticleLayout_t;
018
019    // The type of interpolator to use
020    typedef InterpolatorTag InterpolatorTag_t;
021  };
```

The interpolator tag type is included in the traits class because an application might want the `Particles`-derived to provide the type of interpolator to use. One example of this is the case in which a `gather()` or `scatter()` call occurs in a subroutine which is passed an object of a `Particles`-derived type. This subroutine could extract the desired interpolator type from that object using:

```
// Particles-derived type Particles_t already defined
typedef typename Particles_t::InterpolatorTag_t InterpolatorTag_t;
```

In this short example, this is not really necessary because `InterpolatorTag_t` is being defined and then used within the same file scope. Nevertheless, this illustrates a situation in which the user might want to add new traits to their PTraits class beyond the required traits `AttributeEngineTag_t` and `ParticleLayout_t`.

We can now also define the class which will represent the charged particles in the simulation. As in other examples, this is derived from `Particles`, and templated on a traits class so that such things as its layout and evaluation engine can be quickly, easily, and reliably changed. This class has three intrinsic properties: the particles' positions `R`, their velocities `V`, and their charge/mass ratios `qm`. The class also has a fourth property called `E`, which is used to record the electrical field at each particle's position in order to calculate forces. This calculation will be discussed in greater detail below.

```
024  template <class PT>
025  class ChargedParticles : public Particles<PT>
026  {
027  public:
028    // Typedefs
029    typedef Particles<PT>                          Base_t;
030    typedef typename Base_t::AttributeEngineTag_t  AttributeEngineTag_t;
031    typedef typename Base_t::ParticleLayout_t      ParticleLayout_t;
```

```
032    typedef typename ParticleLayout_t::AxisType_t  AxisType_t;
033    typedef typename ParticleLayout_t::PointType_t PointType_t;
034    typedef typename PT::InterpolatorTag_t         InterpolatorTag_t;
035
036    // Dimensionality
037    static const int dimensions = ParticleLayout_t::dimensions;
038
039    // Constructor: set up layouts, register attributes
040    ChargedParticles(const ParticleLayout_t &pl)
041    : Particles<PT>(pl)
042    {
043      addAttribute(R);
044      addAttribute(V);
045      addAttribute(E);
046      addAttribute(qm);
047    }
048
049    // Position and velocity attributes (as public members)
050    DynamicArray<PointType_t,AttributeEngineTag_t> R;
051    DynamicArray<PointType_t,AttributeEngineTag_t> V;
052    DynamicArray<PointType_t,AttributeEngineTag_t> E;
053    DynamicArray<double,     AttributeEngineTag_t> qm;
054  };
```

With the two classes that the simulation relies upon defined, the program next defines the dependent types, constants, and other values that the application needs. These include the dimensionality of the problem (which can easily be increased to 3), the type of mesh on which the calculations are done, the mesh's size, and so on:

```
058  // Dimensionality of this problem
059  static const int PDim = 2;
060
061  // Engine tag type for attributes
062  typedef MultiPatch<GridTag,Brick> AttrEngineTag_t;
063
064  // Mesh type
065  typedef UniformRectilinearMesh<PDim,Cartesian<PDim>,double> Mesh_t;
066
067  // Centering of Field elements on mesh
068  typedef Cell Centering_t;
069
070  // Geometry type for Fields
071  typedef DiscreteGeometry<Centering_t,Mesh_t> Geometry_t;
072
073  // Field types
074  typedef Field< Geometry_t, double,
075                 MultiPatch<UniformTag,Brick> > DField_t;
076  typedef Field< Geometry_t, Vector<PDim,double>,
077                 MultiPatch<UniformTag,Brick> > VecField_t;
078
079  // Field layout type, derived from Engine type
080  typedef DField_t::Engine_t Engine_t;
081  typedef Engine_t::Layout_t FLayout_t;
082
083  // Type of interpolator
084  typedef NGP InterpolatorTag_t;
085
```

```
086  // Particle traits class
087  typedef PTraits<AttrEngineTag_t,Centering_t,Mesh_t,FLayout_t,
088                  InterpolatorTag_t> PTraits_t;
089
090  // Type of particle layout
091  typedef PTraits_t::ParticleLayout_t PLayout_t;
092
093  // Type of actual particles
094  typedef ChargedParticles<PTraits_t> Particles_t;
095
096  // Grid sizes
097  const int nx = 200, ny = 200;
098
099  // Number of particles in simulation
100  const int NumPart = 400;
101
102  // Number of timesteps in simulation
103  const int NumSteps = 20;
104
105  // The value of pi (some compilers don't define M_PI)
106  const double pi = acos(-1.0);
107
108  // Maximum value for particle q/m ratio
109  const double qmmax = 1.0;
110
111  // Timestep
112  const double dt = 1.0;
```

The preparations above might seem overly elaborate, but the payoff comes when the main simulation routine is written. After the usual initialization call, and the creation of an `Inform` object to handle output, the program defines one geometry object to represent the problem domain, and another that includes a guard layer:

```
115  int main(int argc, char *argv[])
116  {
117    // Initialize POOMA and output stream.
118    Pooma::initialize(argc, argv);
119    Inform out(argv[0]);
120
121    out << "Begin PIC2d example code" << std::endl;
122    out << "------------------------" << std::endl;
123
124    // Create mesh and geometry objects for cell-centered fields.
125    Interval<PDim> meshDomain(nx+1,ny+1);
126    Mesh_t mesh(meshDomain);
127    Geometry_t geometry(mesh);
128
129    // Create a second geometry object that includes a guard layer.
130    GuardLayers<PDim> gl(1);
131    Geometry_t geometryGL(mesh,gl);
```

The program then creates a pair of `Field` objects. The first, `phi`, is a field of `double` values, and records the electrostatic potential at points in the mesh. The second, `EFD`, is a field of two-dimensional `Vector`s, and records the electric field at each mesh point. The types used in these definitions were declared on lines 74-77 above. Note how these definitions are made in terms of other defined types, such as `Geometry_t`, rather than directly in terms of basic types. This allows the application to be modified quickly and reliably with minimal changes to the code.

```
133    // Create field layout objects for our electrostatic potential
134    // and our electric field.  Decomposition is 4 x 4.
135    Loc<PDim> blocks(4,4);
136    FLayout_t flayout(geometry.physicalDomain(),blocks);
137    FLayout_t flayoutGL(geometryGL.physicalDomain(),blocks,gl);
138
139    // Create and initialize electrostatic potential and electric field.
140    DField_t phi(geometryGL,flayoutGL);
141    VecField_t EFD(geometry,flayout);
```

The application now adds periodic boundary conditions to the electrostatic field `phi`, so that particles will not see sharp changes in potential at the edges of the simulation domain. The values of `phi` and `EFD` are then set: `phi` is defined explicitly, while `EFD` records the gradient of `phi`.

```
144    // potential phi = phi0 * sin(2*pi*x/Lx) * cos(4*pi*y/Ly)
145    // Note that phi is a periodic Field
146    // Electric field EFD = -grad(phi);
147    phi.addBoundaryConditions(AllPeriodicFaceBC());
148    double phi0 = 0.01 * static_cast<double>(nx);
149    phi = phi0 * sin(2.0*pi*phi.x().comp(0)/nx)
150               * cos(4.0*pi*phi.x().comp(1)/ny);
151    EFD = -grad<Centering_t>(phi);
```

With the fields in place, the application creates the particles whose motions are to be simulated, and adds periodic boundary conditions to this object as well. The `globalCreate()` call creates the same number of particles on each processor.

```
153    // Create a particle layout object for our use
154    PLayout_t layout(geometry,flayout);
155
156    // Create a Particles object and set periodic boundary conditions
157    Particles_t P(layout);
158    Particles_t::PointType_t lower(0.0,0.0), upper(nx,ny);
159    PeriodicBC<Particles_t::PointType_t> bc(lower,upper);
160    P.addBoundaryCondition(P.R,bc);
161
162    // Create an equal number of particles on each processor
163    // and recompute global domain.
164    P.globalCreate(NumPart);
```

Note that the definitions of `lower` and `upper` could be made dimension-independent by defining them with a loop. If `ng` is an array of `int`s of length `PDim`, then this loop is:

```
Particles_t::PointType_t lower, upper;
for (int d=0; d<PDim; ++d)
{
  lower(d) = 0;
  upper(d) = ng[d];
}
```

The application then randomizes the particles' positions and charge/mass ratios using a sequential loop (since parallel random number generation is not yet in POOMA). Once this has finished, the method `swap()` is called to redistribute the particles based on their positions, i.e., to move each particle to its home processor. The initial positions, velocities, and charge/mass ratios of the particles are then printed out.

```
166    // Random initialization for particle positions in nx by ny domain
```

```
167    // Zero initialization for particle velocities
168    // Random intialization for charge-to-mass ratio from -qmmax to qmmax
169    P.V = Particles_t::PointType_t(0.0);
170    srand(12345U);
171    Particles_t::PointType_t initPos;
172    for (int i = 0; i < NumPart; ++i)
173    {
174      initPos(0) = nx * rand() /
175              static_cast<Particles_t::AxisType_t>(RAND_MAX);
176      initPos(1) = ny * rand() /
177              static_cast<Particles_t::AxisType_t>(RAND_MAX);
178      P.R(i) = initPos;
179      P.qm(i) = (2.0 * rand() / static_cast<double>(RAND_MAX) - 1.0) *
180                qmmax;
181    }
182
183    // Redistribute particle data based on spatial layout
184    P.swap(P.R);
185
186    out << "PIC2d setup complete." << std::endl;
187    out << "----------------------" << std::endl;
188
189    // Display the initial particle positions, velocities and qm values.
190    out << "Initial particle data:" << std::endl;
191    out << "Particle positions: "  << P.R << std::endl;
192    out << "Particle velocities: " << P.V << std::endl;
193    out << "Particle charge-to-mass ratios: " << P.qm << std::endl;
```

The application is finally able to enter its main timestep loop. In each time step, the particles' positions are updated, and then `sync()` is called to invoke boundary conditions, swap particles, and then renumber. A call is then made to `gather()` (line 208) to determine the field at each particle's location. As discussed earlier, this function uses the interpolator to determine values that lie off mesh points. Once the field strength is known, the particles' velocities can be updated:

```
195    // Begin main timestep loop
196    for (int it=1; it <= NumSteps; ++it)
197    {
198      // Advance particle positions
199      out << "Advance particle positions ..." << std::endl;
200      P.R = P.R + dt * P.V;
201
202      // Invoke boundary conditions and update particle distribution
203      out << "Synchronize particles ..." << std::endl;
204      P.sync(P.R);
205
206      // Gather the E field to the particle positions
207      out << "Gather E field ..." << std::endl;
208      gather( P.E, EFD, P.R, Particles_t::InterpolatorTag_t() );
209
210      // Advance the particle velocities
211      out << "Advance particle velocities ..." << std::endl;
212      P.V = P.V + dt * P.qm * P.E;
213    }
```

Finally, the state of the particles at the end of the simulation is printed out, and the simulation is closed down:

```
215    // Display the final particle positions, velocities and qm values.
216    out << "PIC2d timestep loop complete!" << std::endl;
217    out << "-----------------------------" << std::endl;
218    out << "Final particle data:" << std::endl;
219    out << "Particle positions: "  << P.R << std::endl;
220    out << "Particle velocities: " << P.V << std::endl;
221    out << "Particle charge-to-mass ratios: " << P.qm << std::endl;
222
223    // Shut down POOMA and exit
224    out << "End PIC2d example code." << std::endl;
225    out << "-----------------------" << std::endl;
226    Pooma::finalize();
227    return 0;
```

Summary

This tutorial has shown how POOMA's `Field` and `Particles` classes can be combined to create complete physical simulations. While more setup code is required than with Fortran-77 or C, the payoff is high-performance programs that are more flexible and easier to maintain.

[Prev] [Home] [Next]

POOMA Tutorial 11
Text Input and Outupt

Contents:

Introduction
The `Inform` Class
Formatted ASCII Output
The `PrintArray` Class
`dbprint()` and Related Function

Introduction

Standard C++ I/O mechanisms, of course, remain available to POOMA codes. Many POOMA classes have `operator<<()` defined to write an ASCII representation of an instance to a stream. For example,

```
Range<3> r(Range<1>(2,10,2),Range<1>(1,3,1),Range<1>(3));
std::cout << "r = " << r << std::endl;
```

will produce the following output to `stdout`:

```
r = [2:10:2,1:3:1,0:2:1]
```

Classes providing `operator<<()` include

`Array`	`DynamicArray`	`Field`	
`Loc`	`Interval`	`Range`	`IndirectionList`
`Vector`	`Tensor`	`TinyMatrix`	
`UniformGridLayout`	`GridLayout`		
`UniformRectilinearMesh`	`RectilinearMesh`		
`ParticleBCItem`	`UniformLayout`	`SpatialLayout`	

Standard C++ input from `stdin` or files will read values into variables of intrinsic C++ types. The `argc` and `argv` variables work as usual for command-line arguments, except that you should first pass them through `Pooma::initialize()` as described in the tutorial on compiling and running POOMA programs to intercept global POOMA command-line options.

POOMA provides additional enhancements for stream output (the `Inform` class) and for readable, formatted output of large array-like containers (and views of them). The `PrintArray` class manages formatting, and the global `dbprint()` and other `db*()` functions provide convenient shortcuts and a means for printing POOMA container data values interactively from debuggers.

For more serious output and input of data, such as restart files and Field or Array data from large program runs, POOMA provides extensible classes and mechanisms for binary file I/O.

The `Inform` Class

POOMA includes an I/O utility class called `Inform`. This class is basically a smarter `ostream`: as well as printing the values supplied by the programs that use it, it can also format the output to include an optional prefix string, and print out the identifier of the parallel context in which it is used. In addition, it can be used to print messages to multiple output destinations, such as a log file plus standard out.

In normal usage, programs send values to `Inform`s using the overloaded `operator<<`, just as if they were `ostream`s. Each message is assigned the `Inform`'s current level of interest; lower level numbers indicate more important or more interesting messages. Each `Inform` also stores a threshold level internally, and only prints messages whose level numbers are less than or equal to that threshold. The threshold value for an Inform object can be obtained with the `outputLevel()` method; the current level for the next message can be obtained with the `messageLevel()` method. Both methods have associated `set` methods taking integer arguments to modify these values. A quick way to turn off output from an Inform object is to set the output level to a special "off" setting, by calling `setOutputLevel(Inform::off)`.

When running with multiple threads in a context, only one thread does the output, either for standard C++ stream output, or for `Inform` output.. This is the "control" thread, which manages task assignment to the others. It is important to note that any output to an `Inform` which reads data from a multi-patch container is independent of whether other threads might be currently modifying those values. To avoid this, insert a call to `Pooma::blockAndEvaluate()` before the output statement:

```
Array<2,double,MultiPatch<GridTag,Brick> > a(...);
Inform pout;
Pooma::blockAndEvaluate();
pout << "a(23,42) = " << a(23,42) << std::endl;
```

By default, a newly-created `Inform` will only print out messages sent to it on context 0, rather than on all contexts. Programs may change this behavior by calling the method `printContext()`, with the ID of the context on which output is to appear as its argument. If the argument to this method is the constant `Inform::allContexts`, then subsequent messages will be printed on all contexts being used by the program, rather than just one. (Note: currently, POOMA is limited to one context, so this does not yet actually do anything.)

`Inform`s can be constructed in three different ways. The first, and simplest, prints messages to `cout`. By default, output is displayed on context 0, and has no prefix. The other two constructors allow the calling program to specify the file to which output is to be sent, and the mode with which that file is to be opened, or the C++ `ostream` to which output is to be appended:

```
Inform(const char *prefix = 0,
       Context_t context = 0);

Inform(const char *prefix,
       const char *fname,
       int writemode,
       Context_t context = 0);

Inform(const char *prefix,
       std::ostream &outstream,
       Context_t context = 0);
```

Other methods are provided to get and set the prefix to be displayed in front of messages, the `Inform`'s context, the current level of interest of messages, and the threshold for displaying messages. An overloaded set of `open()` methods are also provided to open more output streams within the `Inform`. These methods return an ID which can be used to select particular streams when setting such things as the level of interest. Finally, most of the standard `ostream` manipulators and `operator<<()`s are provided.

Formatted ASCII Output

The POOMA `PrintArray` class has templated `print()` methods that print readable formatted output of large containers of values. It provides methods for controlling formatting parameters like the number of values per line, numeric format, and precision.

The global `dbprint()` template functions are a procedural interface around `PrintArray`, used with a set of global functions for setting common formatting parameters shared by all subsequent `dbprint()` invocations. These are useful shorthand for ASCII output from source code, but more importantly they provide a means to set up nontemplate output functions callable interactively from within a debugger. This is helpful for debugging POOMA programs by examining values from `Array`s and other containers.

The `PrintArray` Class

The typical way to use `PrintArray` is to construct a `PrintArray` object, then use its print() methods for sending formatted ASCII output of POOMA container data to a stream such as `cout` or an `Inform` object. The constructor accepts values for six formatting parameters, which are maintained as member data in the object.

```
PrintArray(int domainWidth = 3, int dataWidth =10,
           int dataPrecision = 4, int carReturn = -1,
           bool scientific = false, int spacing = 1);
```

It has methods to (re)set and get current values for these formatting parameters. The following lists the methods and describes the parameters:

`setDomainWidth(), domainWidth()` :

The output format includes (*base*:*bound*:*stride*,*base*:*bound*:*stride*) prefixes at the beginning of each row of values. This controls the number of columns (digits) to allow for each *base*, *bound*, or *stride* value.

`setDataWidth(), dataWidth()` :

Number of columns per numeric data item. For POOMA multicomponent types such as `Vector` and `Tensor`, this is columns per component.

`setDataPrecision(), dataPrecision()` :

If scientific is true, the number of digits past the decimal point; otherwise, the total number of significant digits.

`setCarReturn(), carReturn()` :

If less than 0, print all values in a row (first array index) one one like of output. If greater than 0, specifies the number of values to print before breaking the output with a carriage return.

`setScientific(), scientific()` :

Whether or not to use scientific notation in output formatting.

`setsSpacing(), spacing()` :

Number of spaces between each data item. For POOMA multicomponent types such as `Vector` and `Tensor`, this is spaces between each whole object.

The `print()` methods of `PrintArray` are member templates:

```
template<class S, class A>
void print(S &s, const A &a) const;

template<class S, class A, class DomainType>
void print(S &s, const A &a, const DomainType &d) const;
```

These take an output stream, a container object, and an optional domain object for explicitly subsetting the container. They work with POOMA `Field`, `Array`, and `DynamicArray` container objects (including attributes from `Particles`), but are not restricted to these. The only restrictions are that the container must export an `enum` value `dimensions`, such as `Array::dimensions`, and must have an array-indexing capability such that `operator()(int io, int i1, ..., int iN)` returns a contained data value. (Here, `N=dimensions-1`.)

If you pass in a view of an `Array`, for example, to the first prototype, the output will show zero-based, unit-stride indexing rather than the original-`Array` indexes specified by the view. To avoid this, use the second prototype and pass in the whole `Array` and a view-subsetting domain object, such as a `Range`, separately. These code snips illustrate the difference, and show what the output is like for a 3D `Array`:

```
Range<3>r(Range<1>(2,10,2),Range<1>(1,3,1),Range<1>(3));
Array<3> a(20,20,20); // ... assign values to a ...

Inform pout;   // An output stream
PrintArray pa; // Use defaults for formatting parameters

pa.print(pout, a(r));
```

prints

```
~~~~~~~~~~~~~~ (0:4:1,0:2:1,0:2:1) ~~~~~~~~~~~~~~


(0:4:1,0:2:1,0):
-------------------------------------------------
(000:004:001,000,000) =        2.5        4.5        6.5        8.5       10.5
(000:004:001,001,000) =        2.5        4.5        6.5        8.5       10.5
(000:004:001,002,000) =        2.5        4.5        6.5        8.5       10.5


(0:4:1,0:2:1,1):
-------------------------------------------------
(000:004:001,000,001) =        2.5        4.5        6.5        8.5       10.5
(000:004:001,001,001) =        2.5        4.5        6.5        8.5       10.5
(000:004:001,002,001) =        2.5        4.5        6.5        8.5       10.5


(0:4:1,0:2:1,2):
-------------------------------------------------
(000:004:001,000,002) =        2.5        4.5        6.5        8.5       10.5
(000:004:001,001,002) =        2.5        4.5        6.5        8.5       10.5
(000:004:001,002,002) =        2.5        4.5        6.5        8.5       10.5
```

while

```
pa.print(pout, a, r);
```

prints

```
~~~~~~~~~~~~~~ (2:10:2,1:3:1,0:2:1) ~~~~~~~~~~~~~~


(2:10:2,1:3:1,0):
-------------------------------------------------
(002:010:002,001,000) =        2.5        4.5        6.5        8.5       10.5
(002:010:002,002,000) =        2.5        4.5        6.5        8.5       10.5
(002:010:002,003,000) =        2.5        4.5        6.5        8.5       10.5


(2:10:2,1:3:1,1):
-------------------------------------------------
(002:010:002,001,001) =        2.5        4.5        6.5        8.5       10.5
(002:010:002,002,001) =        2.5        4.5        6.5        8.5       10.5
```

```
(002:010:002,003,001) =        2.5        4.5        6.5        8.5       10.5


(2:10:2,1:3:1,2):
-----------------------------------------------------
(002:010:002,001,002) =        2.5        4.5        6.5        8.5       10.5
(002:010:002,002,002) =        2.5        4.5        6.5        8.5       10.5
(002:010:002,003,002) =        2.5        4.5        6.5        8.5       10.5
```

`dbprint()` and Related Functions

Many debuggers have a command prompt or expression-evaluation window and allow interactive calling of functions with simple arguments. Few, if any, of these debuggers have a convenient means to invoke template functions even when the templates have been instantiated in the executable code; and none allow interactive construction of objects or invocation of objects' member functions, whether the associated class and/or member functions are templated or not.

Recognizing this, we provide the `dbprint()` function templates, which are a procedural interface to the `PrintArray::print()` member templates:

```
template<class Container>
void dbprint(const Container &c);

template<class Container, class DomainType>
void dbprint(const Container &c, const DomainType &domain);

template<class Container>
void dbprint(const Container &c, const int &i0);

template<class Container>
void dbprint(const Container &c, const int &i0, const int &i1);
// ...

template<class Container>
void dbprint(const Container &c, const int &i0, ..., const int &i20);
```

The first two prototypes map directly to the `PrintArray::print()` functions described in the previous section. The remaining prototypes are for printing single container elements with scalar indexing, and for printing views using sets of integers for base, bound, and stride values in the various dimensions. Prototypes for 1 through 21 integer arguments, skipping {11,13,17,19,20}, allow for "sensible" interpretation of lists of integers as single-element or multi-element views of containers having dimensionality 1 through 7:

dimensions	single element	(base,bound) for each dimension; all stride 1	(base,bound,stride) for each dimension	0:i0-1:1 in each dimension
1	1	2	3	1
2	2	4	6	1
3	3	6	9	1
4	4	8	12	1
5	5	10	15	1
6	6	12	18	1
7	7	14	21	1

Interpretation of various numbers of `int&` *arguments for different dimensionilities.*

You may call any of these functions from your source code, of course, and the compiler will instantiate the appropriate

template instances and underlying `PrintArray::print()` instances. For calling interactively from a debugger, you must make the extra step of adding non-template wrappers for your specific container types, so that the underlying template instances are compiled into your executable. The following code snip illustrates this:

```
// Global typedefs; useful in making user-defined functions below:
const unsigned D = 2;
typedef UniformRectilinearMesh<d> Mesh_t;
typedef Field<DiscreteGeometry<Cell, Mesh_t>, double> ScalarField_t;
typedef Field<DiscreteGeometry<Cell, Mesh_t>, Vector<D> > VectorField_t;
typedef Array<D, double, CompressibleBrick> ScalarArray_t;
typedef Array<D, Vector<D>, CompressibleBrick> VectorArray_t;

class Atoms : public Particles<SharedBrickUniform> {
public:
  // Particle attributes:
  DynamicArray<Vector<D>, SharedBrick> r;
  DynamicArray<Vector<D>, SharedBrick> v;
  // ... rest of class definition ....
  // Constructor: set up layouts, register attributes
  Atoms(const UniformLayout &pl) : Particles<SharedBrickUniform>(pl)
  {
    addAttribute(r);
    addAttribute(v);
  }
};
typedef DynamicArray<Vector<D>, SharedBrick> VAttribute_t;

// User-defined nontemplate dbprint()-type functions:
void sfdbprint(const ScalarField_t &f) { dbprint(f); }
void vfdbprint(const VectorField_t &f) { dbprint(f); }
void sadbprint(const ScalarArray_t &a) { dbprint(a); }
void vadbprint(const VectorArray_t &a) { dbprint(a); }
void pdbprint(const VAttribute_t &pa) { dbprint(pa); }

// Subsetting functions:
// N.B.: these have to have separate names; some debuggers aren't smart enough
// to understand multiple prototypes of function with same name.
void esfdbprint(const ScalarField_t &f, int i) { dbprint(f,i); }
void rsfdbprint(const ScalarField_t &f, int base0, int bound0,
                int stride0, int base1, int bound1, int stride1)
{ dbprint(f, base0,bound0,stride0, base1,bound1,stride1); }
void epdbprint(const VAttribute_t &pa, int i) { dbprint(pa, i); }
void rpdbprint(const VAttribute_t &pa, int base, int bound, int stride)
{ dbprint(pa, base,bound,stride); }

int main(int argc, char* argv[])
{
  // Make and Arrays, and some Fields with GuardLayers<D>(2)
  ScalarField_t s(...);
  VectorField_t v(...);
  ScalarArray_t sa(...);
  VectorArray_t va(...);

  // Make Atoms object:
  Atoms atoms(...);
  atoms.globalCreate(20);
```

```
    //... Assign values to all these...

    // ...Stop the debugger somewhere down here...
```

Note that you must define a separately-named non-template function for each different fully-specified data type you want to examine interactively from the debugger (specified values for all template parameters of `Array` or `Field`, for example). This is a bit cumbersome, but can be worth the trouble.

Following is a screen-shot from an example session running the complete program sketched above, illustrating how to call the user-defined `dbprint()` wrapper functions. Calling syntax may vary from one debugger to the next. In this case, the debugger is `dbx` running on SGI IRIX 6.5. It is stopped at a breakpoint in `main()`:

```
(dbx) ccall dbSetCarReturn(5)
(dbx) ccall sfdbprint(&s)
( -2:004:001, -2) =       -1.5       -0.5        0.5        1.5        2.5
                           3.5        4.5
( -2:004:001, -1) =       -1.5       -0.5        0.5        1.5        2.5
                           3.5        4.5
( -2:004:001,000) =       -1.5       -0.5        0.5        1.5        2.5
                           3.5        4.5
( -2:004:001,001) =       -1.5       -0.5        0.5        1.5        2.5
                           3.5        4.5
( -2:004:001,002) =       -1.5       -0.5        0.5        1.5        2.5
                           3.5        4.5
( -2:004:001,003) =       -1.5       -0.5        0.5        1.5        2.5
                           3.5        4.5
( -2:004:001,004) =       -1.5       -0.5        0.5        1.5        2.5
                           3.5        4.5
(dbx) ccall esfdbprint(&s, 1,1,1)
(000,000) =        0.5
(dbx) ccall rsfdbprint(&s, 1,3,2, 1,2,1)
(001:003:002,001) =        1.5        3.5
(001:003:002,002) =        1.5        3.5
(dbx) ccall dbSetCarReturn(2)
(dbx) ccall pdbprint(&atoms.r)
(000:019:001) = (     1.965,     2.024) (     1.549,     1.807)
                (    0.7699,     2.476) (    0.8934,     2.369)
                (     2.401,     1.617) (    0.7499,     2.148)
                (     0.987,     2.125) (    0.5183,     2.347)
                (     1.792,     1.669) (     1.192,     1.937)
                (    0.9148,     2.503) (     2.114,     1.823)
                (     2.099,      1.19) (     1.855,     1.124)
                (      1.68,    0.4495) (    0.2164,    0.5908)
                (     1.647,     1.128) (      1.36,     1.131)
                (     2.836,     1.302) (   0.06326,    0.4787)
(dbx) ccall epdbprint(&atoms.r, 2)
(002) = (    0.7699,     2.476)
(dbx) ccall rpdbprint(&atoms.r, 2, 6, 2)
(002:006:002) = (    0.7699,     2.476) (     2.401,     1.617)
                (     0.987,     2.125)
(dbx)
```

Note that the first function call, to print the entire `Fields`, includes the global guard layers. Calling `dbprint()` (or `sfdbprint()`) from your source code, you could pass in `s()`, or `s(s.physicalDomain())`, to exclude the global guard layers; this is not possible interactively.

The `dbSetCarReturn()` invocations illustrate more of the POOMA `dB*()` function family. These invoke the corresponding `PrintArray` functions on a global `PrintArray` object maintained internally by POOMA. This sets a

format state that persists from one interactive function call to the next. Here is the set of these functions. Refer to the previous section on `PrintArray` for their meanings:

```
int dbDomainWidth();
void dbSetDomainWidth(int val);
int dbDataWidth();
void dbSetDataWidth(int val);
int dbDataPrecision();
void dbSetDataPrecision(int val);
int dbCarReturn();
void dbSetCarReturn(int val);
bool dbScientific();
void dbSetScientific(bool val);
int dbSpacing();
void dbSetSpacing(int val);
```

Two additional functions allow toggling between the default Inform object used by dbprint() and one or more user-defined Inform objects:

`void dbSetInform(Inform &inform)` :

Replace the default `Inform` object with the input object.

`void dbSwapInform()` :

After a preceding `dbSetInform()`, toggles between the input `Inform` object and the default. That is, repeated calls to `dbSwapInform()` switch back and forth between the two `Inform`s.

[Prev] [Home] [Next]

POOMA Tutorial 12
Object I/O

Contents:

- Overview
- Object Serialization and Object Persistence Models
- Design of POOMA I/O
- What's in POOMA Version 2.2
- Using POOMA I/O
- The *ObjectSet* Interface
  - *ObjectSet* Constructors
  - *ObjectSet::open()*
  - *ObjectSet::flush()* and *ObjectSet::close()*
  - *ObjectSet::store()* and *ObjectSet::retrieve()*
  - Queries on *ObjectSet*
- Data Types Supported in POOMA 2.2
- Use Case
  - *Doof2d* Example Modified for POOMA I/O

Overview

The POOMA framework has been engineered to support rapid development of scientific and engineering applications. POOMA provides its user's with a high-level C++ language interface for creating numerical applications optimized for performance on platforms ranging from desktop computers to parallel supercomputers with thousands of processors. POOMA data abstractions and programming models are general, flexible, and user-extensible.

The POOMA I/O classes have been designed to provide efficient I/O services while keeping to the design philosophy of POOMA. POOMA I/O supports the abstractions that make the POOMA framework powerful and flexible by making the classes that embody them persistent. As with the rest of POOMA, the I/O system is both flexible and extensible by users as well as by developers.

Object Serialization and Object Persistence Models

There are two broad categories of data management appropriate to object-oriented applications. The first is object serialization, and the second is object persistence.

The simplest I/O model is based on inserting data items into input or output streams. Data is typically extracted in the same order as originally stored. Object-oriented applications present special problems for I/O since the the ability of users to add new data types means that many if not most types are unknown to the system. Systems that support object serialization usually have some means of prescribing how the data contained in complex types is to be marshaled and inserted into a stream. Once this definition is in place, new object types can be read or written in the same way as intrinsic types. C++ allows users to overload the insertion operators (<< and >>) for this very purpose. However, as the structure of data types becomes more complicated, the burden falling on users to serialize new types for storage can be quite heavy. Several languages and frameworks provide means of facilitating object serialization. These include, for example, JAVA and Python.

The next level of sophistication in object storage is object persistence. In an object persistence model, objects are stored as a collection of discrete entities, each individually retrievable at random from a collection of objects. A full-featured object-oriented database (OODB) knows enough about the structure of the object types in its collection to perform sophisticated queries based on object metadata.

There is often a tradeoff between these two categories of services. Object serialization is typically more efficient than object database persistence since data is simply marshaled and inserted into a stream. However, the requirement that data-consuming applications know what types of objects to expect as well as their sequence often leads to overly tight coupling between data-producing and data-consuming applications. Thus serialization is fine for monolithic applications performing what amounts to state dumps, but not as good for multi-application collaborative environments. On the other hand, there are many situations when one would just as soon not have the overhead of an object-oriented database no matter how streamlined.

Object-oriented applications benefit enormously from object-oriented data management. After all, the principle reason many programmers prefer object-oriented languages is so that they can create and exploit new data types. Object storage systems provide a way to store and retrieve user-defined types as easily as intrinsic types.

Design of POOMA I/O

The goal of POOMA I/O is to provide object serialization and object database persistence models, both of which have been shown to be extremely useful in object-oriented frameworks. The challenge is to make both of these capabilities flexible enough and lightweight enough to satisfy the requirements of the POOMA framework for extensibility and performance. Here we discuss the basic ideas behind the design of POOMA I/O in order to give the reader a feeling for how these sometimes conflicting requirements can be satisfied simultaneously.

The first level of the design is comprised of a set of classes called the *storage classes* that are transparent to users. They organize any given storage resource into *byte records*. The system does not necessarily know the internal structure of a byte record, only its length in bytes. Records are *elements* of *byte arrays*. Each array is independently accessible within a storage resource and each record or element of a byte array is also independently addressable. A range of elements within a byte array can be read or written in one operation. These byte arrays are automatically extended whenever an operation writes past the current number of elements. Arrays are members of a collection called a *storage set* which serves as the logical interface to storage in terms of arrays. The physical storage in this implementation is a disk file, but a storage set is an

abstraction barrier that need not be associated with a file in general. For example, future implementations may support storage sets based on databases or remote application resources.

The second level is made up of the *object storage classes*. These classes view storage as a set of typed objects called an *object set*. Any instance of a type supported by the I/O system can be stored along with a descriptive label in one operation. Object sets can be queried to reveal the number of objects contained, the types of objects contained, the number of objects of each type, and the labels of each object. A single operation is sufficient to retrieve an object given either its name, or its *instance ID* which is equivalent to its position in the list of object instances for a given type.

The storage of specific types is enabled by specializations of two generic classes: *object serializers* and *object adapters*. As one would infer from the discussion above, serializers serialize objects to a stream, whereas adapters adapt specific types to storage and retrieval in an object set. Adapters often use the services of serializers. The object storage classes in turn use the services provided by the storage set and byte array classes.

To support a different storage type or format, or to optimize I/O for performance, one need only modify the basic storage classes thus leaving the object storage classes unchanged. Several different types of storage can coexist in the same application. The benefit of this design is that new types can be supported simply by creating new serializer and adapter specializations. Our intent is to allow users as well as developers to extend the range of supported types by writing a small amount of new code, or by writing a simple high-level description of the new classes.

The main goal of the POOMA I/O design is to achieve a high level of support for object storage and management without incurring the overhead of a full-featured object-oriented database. Straightforward storage and retrieval operations are provided based on simple queries.

It was also considered important to expose the basic I/O mechanisms through the storage set and byte array classes so that developers could gauge the performance implications of an implementation based on generic storage abstractions. The separation of basic I/O from object management permits performance to be optimized without requiring modifications in any portion of the object management layer.

What's in POOMA Version 2.2

I/O for Version 2.2 of POOMA is experimental. As such it does not support the full scope of capabilities described above, nor the full complement of POOMA framework objects. The reason for including it in this release is to get user feedback and suggestions as early as possible.

Historically, an object persistence model was considered first and object serialization later. The compatibility of these two models, as well as a straightforward solution for supporting and leveraging both, emerged later in design iteration cycles. Thus, in this release users can store and retrieve POOMA objects in an object set, but cannot serialize the same objects to a standard output stream. This feature will be added in the next release.

Since storage adapters are currently hand-crafted, there are only a few basic types supported at this time. Experience gained in writing adapters and serializers for this release will allow us to semi-automate the process of adding support for new types. Some capability of this kind as well as full coverage of all POOMA objects is intended for the next major release of the software.

This release supports standard native binary I/O. Future releases may support storage using the HDF5 format.

Using POOMA I/O

This section describes the basic process of storing and retrieving objects in POOMA. The essential mechanism is very simple. Each supported type may be stored in a collection of objects called an *object set*. An object set is created, opened, and closed like a file. It has three templated member functions to perform object storage and retrieval operations, a *store()* function and two variants of *retrieve()* depending on whether the object is to be recovered by name or by ID. Simple query functions of the object set reveal its contents. Objects can be added to existing objects in an object set, or all objects can be removed upon opening. Once stored, an object cannot be deleted separately. To retrieve an object, the user must supply a default instance of the corresponding type. A typical session in which a user stores data would be as follows:

- Create an object set giving it a name. This may be either a new object set or an existing one to which objects are to be added. To store objects, the access mode must be appropriate for write access.
- Store one or more objects supplying a name or label for each. Names need not be unique.
- Close the object set.

A session in which a user retrieves objects could be described in the following way:

- Create an object set supplying a name matching an existing object set. To retrieve objects the access mode must be appropriate for reading.
- Create default instances of objects matching the ones to be retrieved.
- Retrieve the objects by giving either names or IDs.
- Close the object set.

To aid in retrieving objects, a set of basic queries is provided by the object set interface. The essential functionality of object set queries may be summarized as follows:

- Report the number of distinct types in the object set.
- Report the name of a type given its index *k*, where *k = 0, ... (number of types -1).*
- For a given type indicated by type name or index, report the number of instances.
- For a given type indicated by type name or index *k*, report the name of object *j* where *j = 0 , ..., (number of instances -1).*

The ID of an object is an integer (type *long*) that by convention is the position of the object in the list of instances of that type. That is, if an instance of a given type is second on the list, its ID is 1 (indexed from zero). The primary key for objects contained in an object set is the pair of attributes comprised of its type name or type index and its instance ID. Names are user-defined labels and are not primary keys, i.e., they are not unique and in fact may be null. If a request is made to retrieve an object by name, the object set restores the *first* instance that matches the name.

The following section provides details of the object set interface.

The ObjectSet Interface

The object set interface is the main interface to object storage. To store and retrieve objects, an instance of an object set must exist in the user's application with an access mode appropriate to the intended storage operations.

ObjectSet Constructors

The following constructors create instances of object sets:

ObjectSet() This is the default constructor. Constructed this way, an object set is unusable until an *open()* operation places it in an appropriate state attached to a particular storage resource.

ObjectSet(const std::string& name, StorageResourceType type, StorageAccessMode mode) This is the primary constructor. The arguments are:

name The name of the object set. For file-based storage (the only type for this release) this is literally the name of the file.

type This is an instance of an enumerated type called *StorageResourceType* whose allowed values for this release are:

StdStorage	Standard binary file

mode An instance of an enumerated type called *StorageAccessMode* that defines the access mode. The allowed values are:

storageIn	Read-only access
storageOut	Write-only
storageOutTrunc	Write-only; destroy data if the resource exits
storageInOut	Read-write; append new data to existing data
storageInOutTrunc	Read-write; destroy existing data if the resource exists

Example:

```
ObjectSet obset("DataFile.std", Std5Storage, storageInOutTrunc);
```

Creates an object set *obset* as a binary file whose name will be "DataFile.std." The file is opened for read-write, but if a file by that name already exists, all exisiting data will be destroyed (i.e., the file will be truncated).

ObjectSet::open()

The *open()* operation assumes the existence of an object set and assumes that it has either been default constructed, or that it has been previously closed. There are two variants. They are:

int open(const std::string& name, StorageResourceType type, StorageAccessMode mode) Opens a default object set or closed set assuming all attributes of the object set are new. The arguments have the same meaning as in the main constructor. It returns 0 if successful.

int open(const std::string& name, StorageAccessMode mode) This variant assumes that the

storage resource type has already been set. It generates an error if the object set has only been default constructed, and returns 0 if successful.

Examples:

```
status= obset.open("DataFile.std", storageIn);
assert(status==0);
```

Opens the previous file (assuming it has been closed) in read-only mode.

```
status= obset.open("OtherData.dat", stdStorage, storageOutTrunc);
assert(status==0);
```

Having closed the previous object set, this opens a completely different resource of a different type (standard binary in this case) for output, destroying any pre-existing version.

ObjectSet::flush() and ObjectSet::close()

These functions respectively flush and close the object set. They take no arguments. The *flush()* function ensures that all objects are persistent, and *close()* closes the file or resource. *close()* invokes *flush()* before closing the resource.

ObjectSet::store() and ObjectSet::retrieve()

These functions perform the main storage operations. There are two versions of *retrieve()* depending on whether one wants to retrieve an object by name or by ID.

template <class T>
long store(T& t, const std::string& objectName) Stores an instance of the given type along with a user-defined label. The function returns the object ID assigned by the object set. Valid IDs are zero or greater.

t The given memory-resident object instance.
objectName The user-assigned name or label to be associated with this instance.

template <class T>
int retrieve(T& t, long id) Retrieves an object given its ID. It returns 0 if successful.

t The memory-resident object instance to be instantiated from the persistent version.
id The ID for the stored instance.

template <class T>
int retrieve(T& t, const std::string& objectName) Retrieves an object given its label. Labels are not unique. If there is more than one object of the given type with the same label, it restores the *first* one. It returns 0 if successful.

t The memory-resident object instance to be instantiated from the persistent version.
objectName The user-assigned name or label associated with this instance.

Examples:

```
int nTimeSteps=1000;
long id= obset.store(nTimeSteps, "Number of Time Steps");
assert(id>=0);
```

Stores the given *int* instance with the associated label "Number of Time Steps." An integer

(*long*) ID is returned.

```
int nSteps;
int status= obset.retrieve(nSteps,id);
assert(status==0);
```

Retrieves the value previously stored given the ID, presumably known. Alternatively one could use:

```
status= obset.retrieve(nSteps,"Number of Time Steps");
assert(status==0);
```

Queries on ObjectSet

The following functions allow applications to query the status of an object set:

const std::string& name() const Returns the name of the object set.

StorageAccessMode mode() const Returns the current access mode.

bool isOpen() const Boolean operation to check whether the set is open.

bool isClosed() const Boolean operation to check whether the set is closed.

These functions query the contents of an object set:

int numTypes() const Returns the number of types in the set.

int numInstances(const std::string& typeName) Returns the number of instances of a given type referred to by type name.

typeName The name of the type in question.

int numInstances(long typeID) const Returns the number of instances of a given type referred to by type ID.

typeID The type ID or index. Within a given object set, the types contained are indexed from *0, ..., (number of types -1)*.

const std::string& typeName(long typeID) const Returns the type name given a type ID.

typeID The type ID or index.

long typeID(const std::string& typeName) Returns the type ID given the type name.

typeName The name of the type in question.

const std::string& objectName(const std::string& typeName, long instanceID) Returns the object name given a type name and instance ID.

typeName The name of the type in question.
instanceID The instance of this type. Instances are numbered from *0, ..., (number of instances -1)* for a given type.

const std::string& objectName(long typeID, long instanceID) Returns the object name given the type ID and the instance ID.

typeName The name of the type in question.
instanceID The instance of this type.

Examples:

The following is based on the premise that the application has opened an existing file by creating an instance called *obset* in read-only mode. The application generates a report on the contents of the file.

```
std::string obsetName= obset.name();
int nTypes= obset.numTypes();
std::cout<<"Contents of ObjectSet "<<obsetName<<std::endl;
std::cout<<"Number of types = "<<nTypes<<std::endl;
if(nTypes!=0){
    std::cout<<"Type    Type Name    Number of Instances"<<std::endl;
    int numInstances;
    int j;
    for(int i=0; i<nTypes; i++){
        numInstances= obset.numInstances(i);
        std::cout<<i<<"    "<<obset.typeName(i)<<"    "
                 <<numInstances<<std::endl;
        std::cout<<"    Instance    Object Name"<<std::endl;
        for(j=0; j<numInstances; j++){
            std::cout<<"    "<<j<<"    "
                     <<obset.objectName(i,j)<<std::endl;
        }
        std::cout<<std::endl;
    }
}
```

The next example is based on a similar premise. In this case, the application knows that there are several instances of *complex<double>* called "Field Value." Complex numbers are a templated type in C++ whose conventional type designation in POOMA I/O is "std::complex<T>." The application collects the values by retreiving each instance of this type that matches the name and putting it in a standard C++ vector container.

```
vector<std::complex<double> > fieldVals;
std::complex<double> complexVal;
int nInstances= obset.numInstances("std::complex<T>");
int status;
for(int i=0; i<nInstances; i++){
    if(obset.objectName("std::complex<T>",i)=="Field Value"){
        status= obset.retrieve(complexVal,i);
        assert(status==0);
        fieldVals.push_back(complexVal);
    }
}
```

Data Types Supported in POOMA 2.2

The range of data types supported by the object persistence capability in POOMA Version 2.2 is considerably short of the full scope of POOMA, but basic enough that it should be useful. It should also give a reasonable demonstration of this emerging POOMA framework capability. In the next version, not only with the range of types be considerably broadened, but serialization as well as pesistence will be supported. There will also be tools to facilitate inclusion of new types by users or developers. For now, the following are supported entities:

Intrinsic or atomic data types:

Type	Designation	Description
int	"int"	Native *int*

long	"long"	Native *long*
float	"float"	Native *float*
double	"double"	Native *double*

Complex number instances:

Type	Designation	Description
std::complex<T>	"std::complex<T>"	Complex numbers from the standard numerical library. T may be *float* or *double*.

Standard library strings:

Type	Designation	Description
std::string	"std::string"	Standard string of arbitrary length.

Pooma Vector instances:

Type	Designation	Description
Vector<Dim,T,Engine=Full>	"Vector<Dim,T>"	Pooma Vector class based on the standard Full engine where the dimension *D* may be any size, and *T* is *int*, *long*, *float*, *double*, or *std::complex<T>*.

Pooma Brick and Compressible Brick Arrays:

Type	Designation	Description

Array<Dim,T,Brick> and Array<Dim,T,CompressibleBrick>	"Array<Dim,T,Brick>" and "Array<Dim,T,CompressibleBrick>" respectively	Pooma Array of dimension *Dim=1,... 7* of *Brick* or *CompressibleBrick* engine types. *T* may be *int*, *long*, *float* or *double* in this release.

Pooma Intervals:

Type	Designation	Description
Interval<Dim>	"Interval<Dim>"	Pooma Interval of dimension *Dim=1,... 7.*

Use Case

The following use case demonstrates how object persistence in POOMA Version 2.2 would be used in an application. This is a modification of the *Doof2d* example (simple diffusion calculation) given in another tutorial. The additional I/O instructions are highlighted in italics.

Doof2d Example Modified for POOMA I/O

```
// create arrays
Array<2> a, b;

// create an object set to store the data;
// truncate the file if it already exists
ObjectSet dataSet("Doof2dDB.dat", stdStorage, storageOutTrunc);

// get problem size
int n;
std::cout << "Size (typically 100-1000): ";
std::cin >> n;
int i, niters = n/2;

// create a description for this run using a string stream
// and then store as a string variable
std::ostringstream strstrm;
strstrm<<"This is a run of the Doof2d example with "
    <<" problem size N="<<n<<"."<<std::endl;
strstrm<<"Stencils were not used in this run."<<std::endl;
std::string descr= strstrm.str();
dataSet.store(descr,"Run Description");
```

```
// store the problem size and number of iterations
dataSet.store(n,"Problem Size");
dataSet.store(niters, "Number of Iterations");

// create array domain and resize arrays
Interval<1> N(1,n);
Interval<2> domain(N,N);

// store the problem domain interval
dataSet.store(domain,"Problem Domain Interval");

a.initialize(domain);
b.initialize(domain);

// get domains and constant for diffusion stencil
Interval<1> I(2,n-1), J(2,n-1);
const double fact = 1.0/9.0;

// store the numerical constant factor used to calculate
dataSet.store(fact,"Numerical Factor");

// reset array element values
a = 0.0; b=0.0;
double initialVal= 1000.0;
a(niters,niters) = initialVal;

// store the initial peak value
dataSet.store(initialVal,"Initial Peak Value");

// Run 9pt doof2d without coefficients using expression
std::cout << "Diffusion using expression ..." << std::endl;
std::cout << "iter = 0, a_mid = " << a(niters,niters) << std::endl;
for (i=1; i<=niters; ++i) {
  b(I,J) = fact * (a(I+1,J+1) + a(I+1,J  ) + a(I+1,J-1) +
                   a(I  ,J+1) + a(I  ,J  ) + a(I  ,J-1) +
                   a(I-1,J+1) + a(I-1,J  ) + a(I-1,J-1));
  a = b;
  std::cout << "iter = " << i << ", a_mid = " << a(niters,niters)
            << std::endl;

  // for each iteration store the result array
  // labeled by iteration number
  strstrm.str("");
  strstrm<<"Result at iteration "<<i;
  dataSet.store(a,strstrm.str());
}

dataSet.close();
```

If one were to write and execute the content report generator example given above on this file the output would read:

```
Contents of ObjectSet Doof2dDB.dat
Number of Types=5
Type    Type Name    Number of Instances
0    std::string    1
    Instance    Object Name
    0    Run Description
1    int    2
    Instance    Object Name
    0    Problem Size
    1    Number of Iterations
2    Interval<Dim>    1
    Instance    Object Name
    0    Problem Domain Interval
3    double    2
    Instance    Object Name
    0    Numerical Factor
    1    Initial Peak Value
4    Array<Dim,T,Brick>
(however many iterations)
    Instance    Object Name
    0    Result at Iteration 1
    1    Result at Iteration 2
    2    Result at Iteration 3
    ... (however many iterations)
```

The next example assumes that the application programer has some familiarity with the data-producing application. Let v*isArray(array,string)* be the API to some hypothetical visualization tool that renders false color images of POOMA 2d arrays where *array* is the array and *string* is a standard string label for the plot. The following code segment would take the database file generated by the modified Doof2d example and produce plots.

```
ObjectSet dset("Doof2dDB.dat", stdStorage, storageIn);
int nIters;
int status;
status= dset.retrieve(nIters,"Number of Iterations");
assert(status==0);
std::string plotLabel;
Array<2> array;
for(int i=0;i<nIters;i++){
    plotLabel= dset.objectName("Array<Dim,T,Brick>",i);
    status= dset.retrieve(array,i);
    assert(status==0);
    visArray(array,plotLabel);
}
dset.close()
```

There are several other ways that the data could be recovered assuming less familiarity with the application, and using the object set queries to learn more. More sophisticated queries are needed in order to do a good job of acquiring data when nothing *a priori* is known about the contents of a dataset. Such queries are planned for the next version of POOMA.

[Prev] [Home] [Next]

POOMA Tutorial 13
Compiling, Running, and Debugging POOMA Programs

Contents:

Introduction
Compiling
Compiler Requirements
Installing and Configuring POOMA
How to Decipher Compiler Error Messages
Running
POOMA Runtime Arguments
Object-Based Initialization
Debugging

Introduction

This tutorial includes information on configuring and building the POOMA library, and application programs using POOMA. It also discusses some topics about running POOMA programs, and gives some anecdotal information about debuggers and debugging.

Compiling

Compiler Requirements

POOMA has been extensively tested with the following C++ compilers:

- KAI C++ 3.3e or higher
  Kuck and Associates (http://www.kai.com)
  Most UNIX platforms, including Linux
- EGCS/GCC (snapshot after 5/15/99)
  GCC Home Page (http://gcc.gnu.org)
  Most UNIX platforms, including Linux
- CodeWarrior Professional 5
  Metrowerks (http://www.metrowerks.com)
  Macintosh, Windows 95/98/NT
- Intel C++ (part of VTune 4.0)
  Intel (http://support.intel.com/support/performancetools/vtune)
  Windows 95/98/NT

Other compilers based on version 2.38 or later of the Edison Design Group (EDG) front-end may also be able to compile POOMA. However, we know that CFront-based compilers, Visual C++ 5.0/6.0, and GNU C++ 2.91 do not support the ISO features necessary to compile POOMA. These features include:

- Default template arguments
- Partial ordering of function templates
- Member templates
- Explicit instantiation of templates
- ANSI/ISO keywords like `mutable` and `typename`
- ANSI/ISO types like `bool` and a templated complex number class
- ANSI/ISO casts like `const_cast`, `static_cast`, `dynamic_cast`, and `reinterpret_cast`
- RTTI (runtime type identification)
- Namespaces
- Exceptions

In addition, POOMA assumes a fairly standard C++ library including:

- I/O streams
- Standard template library
- Numerics
- Strings

Ideally, the library should support ISO standard ".h-less" headers like `<vector>` that place definitions in the `std::` namespace.

The compiler features listed above are an absolute requirement. For example, if your compiler does not support member templates, there is no amount of work that will get POOMA to compile. If your compiler has limited support for these features, it is worth trying to build the library, but there are no assurances of success. Minor deficiencies in the libraries can also be worked around. In fact, we have had to use a mixture of .h-less and .h headers in order to work around various compiler/library problems:

- .h versions of C headers (e.g., `math.h`, *not* `cmath`)
  reason: Intel/Microsoft platform doesn't have a `cmath`.
- .h version of `complex.h`
  reason: complex math functions must be in same namespace as double/float versions for PETE (POOMA's expression template engine) to work

POOMA-based programs can use either .h or non .h headers for everything else in the standard C++ library, but should of course be consistent.

Installing and Configuring POOMA

The POOMA build system can handle several different configurations at once; this allows developers to building `libpooma.a` and POOMA applications for several configurations at the same time. Each configuration is referred to as a *suite*, and is described by a *suite file*. This is the main file that the configuration script described below sets up when it runs.

Note: this configuration script is presently available only for Unix platforms. Programmers who are installing POOMA on the Apple Macintosh, or under Microsoft Windows will need to change `#define` definitions manually. Under CodeWarrior, on both Macintosh and Windows, these definitions are in the file:

```
pooma-2.2.0/src/arch/Metrowerks/Pooma.prefix.h
```

If the Intel VTune C++ compiler is being used with Microsoft Visual C++ on Windows, the definitions that need to be changed are in:

```
pooma-2.2.0/src/arch/Intel/PoomaConfigurations.h
```

In addition, when the configuration script is run to set up a new suite, it will go through all the subdirectories and create a directory called `<suite>` in each subdirectory that contains files that will be compiled (where `<suite>` is the name of the suite). As described below, the environment variable `POOMASUITE` must be set to the value of the suite to compile after the configuration script is run.

The POOMA configuration script is located in the top level of the POOMA distribution tree, and is called `configure`. It is written in Perl, and does the following;

- Creates a suite file called `config/<suite>.suite.mk` (where `<suite>` is the actual name of the suite) that has settings for building POOMA. The suite file is included by the other makefiles in POOMA to get names of, and arguments for, the compiler and linker. After a suite file has been generated using `configure`, developers set the environment variable `POOMASUITE` to the name of that suite, and run make. It is a good idea to set the variable `TMPDIR` to the name of a temporary directory at this time as well. If this is not done, the POOMA build system will use `/tmp/$POOMASUITE`, which might cause conflicts if two or more builds are being done simultaneously.
- Creates a library build directory called `lib/<suite>`, and puts several files in it:
  - a makefile
  - a "stub" makefile, for use in installation
  - a `PoomaConfiguration.h` file, with `#define` statements indicating how to build POOMA. Almost all `#defines` take one of the following forms:

```
#define SOME_POOMA_VARIABLE    POOMA_YES
#define ANOTHER_POOMA_VARIABLE POOMA_NO
```

- Does other small setup tasks so that POOMA can be built with a new suite.

The most useful arguments for `configure` are:

`--arch <arch>`

Specify an architecture to configure for. The directory `config/arch` has several files with `.conf` extensions, one for each combination of machine and compiler that the current version of POOMA supports. The recommended procedure is to select an architecture file, edit it if necessary, and then run `configure --arch <arch>` (plus any other options) from the top directory of the POOMA distribution. If the `POOMASUITE` environment variable is set, and you do not use the `--arch` flag, the value of `POOMASUITE` will be used instead. One or the other of these methods must be used to specify the desired suite.

`--suite <suite>`

Specify the name of the suite file and `lib/<suite>` build directory to create. This can be different than `<arch>`, but if you do not give the `--suite` option, `<arch>` will be used. If the `--suite` flag is not given then:

- if `POOMASUITE` is set, that will be used for `<suite>`
- if `POOMASUITE` is not set, `<arch>` will be used.

`--prefix <installdir>`

This selects where to install POOMA after you have built the library. The `INSTALL` file (located in the root directory of the POOMA distribution, along with the `LICENSE` file) describes the directory tree that gets created during installation.

`--opt` or `--debug`

Select whether to build optimized or debug library by default.

`--preinst` or `--nopreinst`

If `--preinst` is used, the library will pre-instantiate versions of several classes for several types and dimensions. This step is not necessary; the library will build more quickly if you do not use it, but applications may build more quickly.

`--ex` or `--noex`

Enable or disable exception handling. Some compilers produce more efficient code and compile faster when exceptions are turned off.

`--parallel` or `--serial`

These flags determine whether POOMA will use SMARTS or not. If `--parallel` is given, the SMARTS header files will be included and parallel evaluation will be done. Otherwise, all operations will run in serial.
SMARTS is the thread and dataflow package that POOMA uses for the multithreaded operation. It was also developed at the Advanced Computing Laboratory, and is available on the same CD-ROM as POOMA. This release of SMARTS only runs on Unix platforms; in order to use SMARTS with POOMA, you must:

- compile and install SMARTS before compiling POOMA; and

- set the SMARTSDIR environment variable to the installation directory for SMARTS after installing SMARTS, but before running POOMA's `configure` script.

`--v`

This flag causes compilers and linkers to print very verbose output.

The mode table in an earlier tutorial summarizes the modes produced by different combinations of configuration flags.

The `configure` scripts also has options that will add extra `-I`, `-D`, `-L` and other flags to your compilations. Run `configure -h` to see a complete list of options.

After running `configure` and creating a suite file, set the environment variable `POOMASUITE` to the name of the suite, and run make. There is a makefile at the top level of the POOMA distribution, and in the `lib/<suite>` directory. When make is finished, there should be a `lib/<suite>/libpooma.a` library file.

This file can be used in one of two ways. Programmers who are extending POOMA can use `libpooma.a` without any other work, since the library may need to be recompiled often. Such developers can build the programs in the `benchmarks` and `examples` directories (such as `examples/Doof2d`) by running make in those directories.

Programmers who are just using the library will have to install it. Typing `make install` will install `libpooma.a` and the necessary source files in the `<installdir>` directory (specified with the `--prefix` flag to `configure`). Users should then set the environment variable `POOMADIR` to `<installdir>`, and `POOMAARCH` to a string indicating the type of build architecture. This is *not* the same as the `<arch>` name used for configure, but is instead just a string indicating the type of machine, and for this release may be one of `sgi64`, `sgin32`, `sgi32`, or `linux`.

After doing all of this, users can go into any subdirectory under `examples` or `benchmarks` and run:

```
make -f Makefile.user
```

`Makefile.user` first includes a "stub" makefile from the directory where `libpooma.a` is installed. This stub makefile contains settings such as `POOMA_INCLUDES` that are needed to build POOMA applications. For example, the `Makefile.user` file in `examples/Doof2d` contains the following:

```
### include the POOMA makefile stub, to get compiler flags and libraries
include $(POOMADIR)/$(POOMAARCH)/lib/Makefile.pooma

### the name of the example code to compile
EXAMPLE = Doof2d

### the main target for this makefile
$(EXAMPLE): $(EXAMPLE).cpp
        $(POOMA_CXX) $(POOMA_CXX_DBG_ARGS) -o $(EXAMPLE) $(EXAMPLE).cpp
$(POOMA_INCLUDES) $(POOMA_DEFINES) $(POOMA_LIBS)
```

How to Decipher Compiler Error Messages

POOMA makes extensive use of templates to achieve high performance. Unfortunately, this means that a simple mistake often results in dozens of compiler error messages that are both long and obscure. These messages are often tough for experienced C++ programmers to interpret and can be downright scary for newcomers to the language. There is no simple formula for dealing with these messages, but there are strategies that can reduce the pain associated with the process.

To begin with, consider the program below:

```
01  #include "Pooma/Arrays.h"
02
03  int main(int argc, char *argv[])
04  {
05      Pooma::initialize();
06
07      int p, *pp = &p;
```

```
08      Array<1> z(6);
09      for (p = 0; p < 6; p++)
10          z(PP) = p;
11
12      Pooma::finalize();
13
14      return 0;
15  }
```

KCC 3.3 reports the following impressive set of error messages (please be patient while this scrolls past you):

```
"src/Array/Array.h", line 416: error: name followed by "::" must be a class or
          namespace name
        CTAssert(SDomain_t::dimensions == dimensions);
        ^
          detected during instantiation of "ArrayViewReturn<ConstArray<Dim, T,
                    Engine>::Engine_t, TemporaryNewDomain1<ConstArray<Dim, T,
                    Engine>::Domain_t, Sub1>::SliceType_t>::Type_t Array<Dim,
                    T, EngineTag>::operator()(const Sub1 &) const [with Dim=1,
                    T=double, EngineTag=Brick, Sub1=int *]" at line 10 of
                    "test.cpp"

"src/Array/Array.h", line 416: error: class "PoomaCTAssert<<error-constant>>"
          has no member "test"
        CTAssert(SDomain_t::dimensions == dimensions);
        ^
          detected during instantiation of "ArrayViewReturn<ConstArray<Dim, T,
                    Engine>::Engine_t, TemporaryNewDomain1<ConstArray<Dim, T,
                    Engine>::Domain_t, Sub1>::SliceType_t>::Type_t Array<Dim,
                    T, EngineTag>::operator()(const Sub1 &) const [with Dim=1,
                    T=double, EngineTag=Brick, Sub1=int *]" at line 10 of
                    "test.cpp"

"src/Domain/DomainTraits.Loc.h", line 190: error: a value of type
          "DomainTraitsScalar<int *, int *>::Element_t" cannot be assigned to
          an entity of type "DomainTraits<Loc<1>>::Storage_t"
      dom = DomainTraits<T>::getFirst(newdom);
          ^
          detected during:
            instantiation of "void
                      DomainTraits<Loc<1>>::setDomain(DomainTraits<Loc<1>>::Sto
                      rage_t &, const T &) [with T=int *]" at line 286 of
                      "src/Domain/Domain.h"
            instantiation of "void SetDomainFunctor<DT, ST, T, UT,
                      wildcard>::setDomain(ST &, const T &) [with
                      DT=DomainTraits<Loc<1>>,
                      ST=DomainBase<DomainTraits<Loc<1>>>::Storage_t,
                      T=DomainTraitsScalar<int *, int *>::PointDomain_t,
                      UT=DomainTraitsScalar<int *, int *>::PointDomain_t,
                      wildcard=false]" at line 395 of "src/Domain/Domain.h"
            instantiation of "void Domain<1, DT>::setDomain(const T &) [with
                      DT=DomainTraits<Loc<1>>, T=DomainTraitsScalar<int *, int
                      *>::PointDomain_t]" at line 234 of "src/Domain/Loc.h"
            instantiation of "void CopyLocStorageImpl<Dim, T, 1,
                      false>::copy(Loc<Dim> &, const T &) [with Dim=1, T=int
                      *]" at line 242 of "src/Domain/Loc.h"
            instantiation of "void CopyLocStorage<Dim, T>::copy(Loc<Dim> &,
```

```
                    const T &) [with Dim=1, T=int *]" at line 410 of
                    "src/Domain/Loc.h"
          instantiation of "Loc<1>::Loc(const T1 &) [with T1=int *]" at line
                    78 of "src/Array/Array.h"
          instantiation of "ArrayViewReturn2<Engine, Domain, 1>::Type_t
                    ArrayViewReturn2<Engine, Domain, 1>::eval(const Engine
                    &, const Domain &) [with Engine=Engine<1, double,
                    Brick>, Domain=int *]" at line 89 of "src/Array/Array.h"
          instantiation of "ArrayViewReturn<Engine, Domain>::Type_t
                    ArrayViewReturn<Engine, Domain>::eval(const Engine &,
                    const Domain &) [with Engine=Engine<1, double, Brick>,
                    Domain=int *]" at line 418 of "src/Array/Array.h"
          instantiation of "ArrayViewReturn<ConstArray<Dim, T,
                    Engine>::Engine_t, TemporaryNewDomain1<ConstArray<Dim,
                    T, Engine>::Domain_t, Sub1>::SliceType_t>::Type_t
                    Array<Dim, T, EngineTag>::operator()(const Sub1 &) const
                    [with Dim=1, T=double, EngineTag=Brick, Sub1=int *]" at
                    line 10 of
                    "test.cpp"


"src/Domain/NewDomain.h", line 753: error: name followed by "::" must be a
          class or namespace name
        SliceType_t retval = AllDomain<SliceType_t::dimensions>();
                                       ^
          detected during:
            instantiation of "NewDomain1<T1>::SliceType_t
                    NewDomain1<T1>::combineSlice(const UT &, const T1 &)
                    [with T1=int *, UT=Interval<1>]" at line 128 of
                    "src/Array/ConstArray.h"
            instantiation of "TemporaryNewDomain1<Domain, Sub>::SliceType_t
                    TemporaryNewDomain1<Domain, Sub>::combineSlice(const
                    Domain &, const Sub &) [with Domain=Interval<1>, Sub=int
                    *]" at line 419 of "src/Array/Array.h"
            instantiation of "ArrayViewReturn<ConstArray<Dim, T,
                    Engine>::Engine_t, TemporaryNewDomain1<ConstArray<Dim,
                    T, Engine>::Domain_t, Sub1>::SliceType_t>::Type_t
                    Array<Dim, T, EngineTag>::operator()(const Sub1 &) const
                    [with Dim=1, T=double, EngineTag=Brick, Sub1=int *]" at
                    line 10 of
                    "test.cpp"

"src/Domain/AllDomain.h", line 84: error: class
          "PoomaCTAssert<<error-constant>>" has no member "test"
      CTAssert(Dim > 0);
      ^
          detected during:
            instantiation of
                    "AllDomain<Dim>::AllDomain() [with Dim=<error-constant>]"
                    at line 753 of "src/Domain/NewDomain.h"
            instantiation of "NewDomain1<T1>::SliceType_t
                    NewDomain1<T1>::combineSlice(const UT &, const T1 &)
                    [with T1=int *, UT=Interval<1>]" at line 128 of
                    "src/Array/ConstArray.h"
            instantiation of "TemporaryNewDomain1<Domain, Sub>::SliceType_t
                    TemporaryNewDomain1<Domain, Sub>::combineSlice(const
                    Domain &, const Sub &) [with Domain=Interval<1>, Sub=int
```

```
                    *]" at line 419 of "src/Array/Array.h"
          instantiation of "ArrayViewReturn<ConstArray<Dim, T,
                    Engine>::Engine_t, TemporaryNewDomain1<ConstArray<Dim,
                    T, Engine>::Domain_t, Sub1>::SliceType_t>::Type_t
                    Array<Dim, T, EngineTag>::operator()(const Sub1 &) const
                    [with Dim=1, T=double, EngineTag=Brick, Sub1=int *]" at
                    line 10 of
                    "test.cpp"

"src/Domain/NewDomain.h", line 753: error: no suitable conversion function
          from "AllDomain<<error-constant>>" to
          "NewDomain1<int *>::SliceType_t" exists
        SliceType_t retval = AllDomain<SliceType_t::dimensions>();
                             ^
          detected during:
            instantiation of "NewDomain1<T1>::SliceType_t
                    NewDomain1<T1>::combineSlice(const UT &, const T1 &)
                    [with T1=int *, UT=Interval<1>]" at line 128 of
                    "src/Array/ConstArray.h"
            instantiation of "TemporaryNewDomain1<Domain, Sub>::SliceType_t
                    TemporaryNewDomain1<Domain, Sub>::combineSlice(const
                    Domain &, const Sub &) [with Domain=Interval<1>, Sub=int
                    *]" at line 419 of "src/Array/Array.h"
            instantiation of "ArrayViewReturn<ConstArray<Dim, T,
                    Engine>::Engine_t, TemporaryNewDomain1<ConstArray<Dim,
                    T, Engine>::Domain_t, Sub1>::SliceType_t>::Type_t
                    Array<Dim, T, EngineTag>::operator()(const Sub1 &) const
                    [with Dim=1, T=double, EngineTag=Brick, Sub1=int *]" at
                    line 10 of
                    "test.cpp"

"src/Domain/NewDomain.h", line 131: error: expression must have class type
        DomainTraits<RT>::getDomain(rt, DS + i).setDomain(
        ^
          detected during:
            instantiation of "void CombineSliceDomainWC<RT, UT, CT, DS,
                    SliceDS, incl, wc>::combine(RT &, const UT &, const CT
                    &) [with RT=NewDomain1<int *>::SliceType_t,
                    UT=Interval<1>, CT=int *, DS=0, SliceDS=0, incl=false,
                    wc=false]" at line 207
            instantiation of "void CombineSliceDomain<RT, UT, CT, DS, SliceDS,
                    incl>::combine(RT &, const UT &, const CT &) [with
                    RT=NewDomain1<int *>::SliceType_t, UT=Interval<1>,
                    CT=int *, DS=0, SliceDS=0, incl=false]" at line 766
            instantiation of "RT &NewDomain1<T1>::fillSlice(RT &, const UT &,
                    const T1 &) [with T1=int *, RT=NewDomain1<int
                    *>::SliceType_t, UT=Interval<1>]" at line 754
            instantiation of "NewDomain1<T1>::SliceType_t
                    NewDomain1<T1>::combineSlice(const UT &, const T1 &)
                    [with T1=int *, UT=Interval<1>]" at line 128 of
                    "src/Array/ConstArray.h"
            instantiation of "TemporaryNewDomain1<Domain, Sub>::SliceType_t
                    TemporaryNewDomain1<Domain, Sub>::combineSlice(const
                    Domain &, const Sub &) [with Domain=Interval<1>, Sub=int
                    *]" at line 419 of "src/Array/Array.h"
            instantiation of "ArrayViewReturn<ConstArray<Dim, T,
                    Engine>::Engine_t, TemporaryNewDomain1<ConstArray<Dim,
```

```
T, Engine>::Domain_t, Sub1>::SliceType_t>::Type_t
Array<Dim, T, EngineTag>::operator()(const Sub1 &) const
[with Dim=1, T=double, EngineTag=Brick, Sub1=int *]" at
line 10 of
"test.cpp"
```

The first thing to keep in mind is that the error messages are telling you exactly what went wrong in your program. However, like a patient speaking to a doctor, the compiler is reporting symptoms: "It hurts, here, here, and here." It isn't saying directly, "You have accidentally used an `int*` to index array `z`."

Second, start at the first message and work down. As you can see, C++ compilers will often report several different error messages for the same mistake. The first one is usually the most direct statement of what's wrong so start there. In our example, KCC is reporting an error at line 416 of the POOMA header `Array/Array.h`. This line reads:

```
CTAssert(SDomain_t::dimensions == dimensions);
```

It is specifically complaining about the fact that it doesn't think `SDomain_t` is a class or namespace name, which means that qualifying it with '`::`' doesn't make sense. This is a symptom, but it isn't very useful, especially to someone who isn't familiar with the innards of POOMA. It may be disconcerting that the error message is in POOMA code. However, it is simply the reality with templates that bad user code can result in a template error deep inside POOMA.

Third, the real information is in the instantiation chain. By "instantiation chain", we mean the set of templates, starting with your code, that the C++ compiler was instantiating when it ran into trouble. In KCC errors, the instantiation chain can be recognized by a series of lines beginning with "detected during: instantiation of". The best way to read these chains is from the instantiation closest to user code to that deepest in POOMA. In the first error message, this is easy because there is only one instantiation listed. KCC claims it trying to instantiate:

```
Array<Dim, T, Engine>::operator()(const Sub &)
```

where: `Dim` is 1, `T` is `double`, `EngineTag` is `Brick`, and `Sub1` is `int*`, at line 10 of our example program. Now, this is useful because we see that the problem is with the line:

```
z(PP) = p;
```

There is indeed a call to `operator()` call on that line, i.e. `z(PP)`. Moreover, KCC is telling us that the argument we passed in has a type `int*`, which is not a legal domain type. If we change `z(PP)` to `z(p)`, the problem is solved.

The other error messages give essentially the same information in different ways. More complicated situations may require following the instantiation chain through several levels. The most important thing is not to get blinded by the quantity of output.

The error messages produced by EGCS are formatted differently, but the procedure for interpreting them is the same. Unfortunately, CodeWarrior Professional 4 does not print out an instantiation chain, which makes diagnosing template problems very difficult. Metrowerks knows about this problem and is fixing it. However, until then, we can only suggest compiling your code with EGCS or KCC as a means to diagnose difficult problems.

Running

POOMA Runtime Arguments

The following run-time flags can be used to control various aspects of the behavior of a POOMA-based application:

- `--pooma-debug` *N*: set the debug output level to *N*.
- `--pooma-blocking-expressions`: force POOMA to block after every data-parallel statement.
- `--pooma-threads` *N*: explicitly set the number of threads to be used (i.e. the degree of concurrency).
- `--pooma-help`: print out a summary help message showing the available flags.

The set of flags shown below control log messages, warnings, debugging output, and so on. All of these options have a `-no` form as well, such as `--pooma-noinfo`.

- `--pooma-info`: print info messages.
- `--pooma-warn`: print warning messages.
- `--pooma-err`: print error messages.
- `--pooma-log` *`file`*: log output to file.
- `--pooma-stats`: print runtime statistics at end of execution.

The first four of the flags above are related to a set of macros defined in the `src/Pooma/Pooma.h` header file. The first of these, `POOMA_PRINT(stream,text)`, prints a message to a given stream in a thread-safe manner. The second, `POOMA_PRINTDEBUG(level,text)`, prints a message to the POOMA debug output stream `POOMA::pdebug` if the `POOMA_PRINTDEBUG` option was selected when POOMA was built. The last three macros are `POOMA_INFO(text)`, `POOMA_WARN(text)`, and `POOMA_ERROR(text)`, which print messages to the information, warning, and error output streams (`Pooma::pinfo`, `Pooma::pwarn`, and `Pooma::perr` respectively). These four streams are actually predefined `Inform` objects (described in the tutorial on Text I/O).

There are also flags that globally affect certain POOMA classes:

- `--pooma-nocompress`: disable compression of compressible bricks.
- `--pooma-nodeferred-guardfills`: disable deferred filling of guards.

Finally, these three flags are used by POOMA's SMARTS threading package, and are not fully implemented in this release:

- `--pooma-smarts-hardinit`: memory allocation will respect hardware affinity.
- `--pooma-smarts-hardrun`: tasks will only be run by threads with the correct hardware affinity.
- `--pooma-smarts-lockthreads`: the operating system will not migrate threads.

Object-Based Initialization

As mentioned in the first tutorial, POOMA can be initialized by passing `argc` and `argv` to `Pooma::initialize()`, or by creating an instance of `Pooma::Options`, configuring it, and then passing that options object to `Pooma::initialize()`. Thus, instead of using:

```
Pooma::initialize(argc, argv);
```

a program can do the following:

```
Pooma::Options opts;                    // create the options object
opts.concurrency(8);                    // tell POOMA to use 8 threads
opts.logfile("pooma.log");              // turn on output logging
Pooma::initialize(opts);                // initialize Pooma
```

These two methods can be combined, which allows a program to override any options the user might have specified:

```
Pooma::Options opts(argc, argv);        // parse command line
opts.concurrency(8);                    // but always use 8 threads
Pooma::initialize(opts);                // actual initialization
```

For more information on the configuration options available to POOMA programs, please see the POOMA documentation.

Debugging

Debugging the templated classes and functions in POOMA codes is challenging. Many debuggers have difficulty with finding and stopping in the particular template instance you're interested in. Few, if any, debuggers allow invocation of member functions from objects, whether they are instances of template or nontemplate classes.

Future revisions of this tutorial may include more information on debugging. For now, we include some anecdotal information that may be helpful:

The Metrowerks CodeWarrior Professional 5.2 debugger does a good job of understanding template code, and correctly demangling symbol names. The Windows version is much less successful than the Macintosh version in maintaining proper

state when you trace into template functions and template or nontemplate members of template classes. On Windows, it often fails to recognize any local variables.

On IRIX 6.5, we have had some success with `dbx`, `TotalView` (TotalView 3.9 or higher, Etnus (http://www.etnus.com)), and SGI's `cvd` debuggers. The following table indicates combinations of compilers and debuggers on IRIX which are compatible:

	TotalView	**dbx**	**cvd**
KCC	*compatible*	*incompatible*	*incompatible*
CC	*compatible*	*compatible*	*compatible*
EGCS (g++)	*incompatible*	*compatible*	*compatible**

Compatible compiler/debugger combinations on IRIX 6.5.
**Object member access sometimes crashes cvd with EGCS.*

These debuggers fail in some cases to demangle names of objects which are instances of template classes.

See also the discussion of the `dbprint()` function family. This describes how to set up function prototypes allowing you to examine data values from POOMA containers like `Field` and `Array` interactively from some debuggers.

[Prev] [Home] [Next]

POOMA Tutorials
A Quick Self-Test

Contents:

- Introduction
- Questions
  - Object Creation
  - Virtual Methods and Inheritance
  - Trait Classes
  - Values, References, and Constant References
- Answers
  - Object Creation
  - Virtual Methods and Inheritance
  - Trait Classes
  - Values, References, and Constant References

Introduction

The implementation of POOMA uses many advanced or obscure features of the ANSI/ISO C++ standard. Its interface is less exacting, but programmers must still have a solid understanding of C++ to use it effectively. If you feel comfortable with the questions below, and their answers, you should have little or no difficulty using POOMA. If, on the other hand, you find the questions and their answers difficult, you may wish to look at some of the books in the recommended reading before trying to use this library.

Questions

Object Creation

Assume that a default constructor, a copy constructor, and an overloaded assignment operator have been defined for the class `Fred`. How many times is each called when the following program is executed?

```
Fred red;

Fred func(
    Fred cyan
){
    Fred magenta;
    magenta = cyan;
    return magenta;
}

int main()
{
    Fred green = red;
```

```
    blue = func(green);
    return 0;
}
```

Virtual Methods and Inheritance

What does the following program print out?

```
#include <iostream>
#include <iomanip>
using namespace std;

class A
{
  public :
    A()                 { cout << "A new" << endl; }
    virtual void left() { cout << "A left" << endl; }
    void right()        { cout << "A right" << endl; }
};

class B : public A
{
  public :
    B()                 { cout << "B new" << endl; }
    void left()         { cout << "B left" << endl; }
    void right()        { cout << "B right" << endl; }
};

int main()
{
    A a;
    a.left();
    a.right();
    cout << endl;

    B b;
    b.left();
    b.right();
    cout << endl;

    A * ap = &b;
    ap->left();
    ap->right();
    cout << endl;

    A * ap = (A*)&b;
    ap->left();
    ap->right();
    cout << endl;

    ap->A::left();
    ((A*)ap)->left();
    ((A*)ap)->right();

    return 0;
}
```

Trait Classes

What does the following program print out?

```
class Blue
{
  public :
    enum { Val = 240; };
};

template<class T>
class Green
{
  public :
    const int Val = 88;
};

template<class T>
class Red
{
  public :
    enum { Val = T::Val/2; };
};

int main()
{
    cout << Blue::Val << endl;
    cout << Green<Blue>::Val << endl;
    cout << Red<Blue>::Val << endl;
    cout << Red<Green<Blue>>::Val << endl;
    cout << Red<Green<Green<Blue>>>::Val << endl;
    cout << Red<Red<Green<Blue>>>::Val << endl;
    return 0;
}
```

Values, References, and Constant References

Which of the calls to `value()`, `reference`, and `const_reference` below produce errors during compilation?

```
void value(int x)
{}

void reference(int & x)
{}

void const_reference(const int & x)
{}

int main()
{
    int x;
    const int y = 2;

    value(1);
    value(x);
    value(y);
    value(x+1);
```

```
    reference(1);
    reference(x);
    reference(y);
    reference(x+1);

    const_reference(1);
    const_reference(x);
    const_reference(y);
    const_reference(x+1);

    return 0;
}
```

Answers

Object Creation

The listing below shows where constructor calls and assignments occur:

```
Fred red;                           // default constructor

Fred func(
    Fred cyan                       // copy constructor
                                    // (pass by value)
){
    Fred magenta;                   // default constructor
    magenta = cyan;                 // assignment operator
    return magenta;                 // copy constructor
                                    // (magenta is copied into
                                    // a nameless temporary to
                                    // be returned)
}

int main()
{
    Fred green = red;               // copy constructor
    blue = func(green);             // copy constructor twice
                                    // ('green' is copied into
                                    // 'cyan' during call, and
                                    // temporary return value
                                    // is copied into 'blue' on
                                    // exit)
    return 0;
}
```

Virtual Methods and Inheritance

The program prints the following:

```
A new                               // A::A()
A left                              // A::left()
A right                             // A::right()

A new                               // B::B() invokes A::A()
B new                               // body of B::B()
```

```
B left                                  // B::left()
B right                                 // B::right()

B left                                  // left() is virtual
A right                                 // right() is not virtual

B left                                  // cast on right irrelevant
A right                                 // right() is not virtual

A left                                  // exact method named
B left                                  // cast on left irrelevant
A right                                 // right() is not virtual
```

Trait Classes

The key here is that `Green` always defines its own `Val`, while `Red` defines its `Val` in terms of its argument class's `Val`. The answer is therefore:

```
int main()
{
    cout << Blue::Val << endl;                    // 240
    cout << Green<Blue>::Val << endl;             // 88
    cout << Red<Blue>::Val << endl;               // 120
    cout << Red<Green<Blue>>::Val << endl;        // 44
    cout << Red<Green<Green<Blue>>>::Val << endl; // 44
    cout << Red<Red<Green<Blue>>>::Val << endl;   // 22
    return 0;
}
```

Values, References, and Constant References

The only outright errors occur when a constant value (such as a literal or the result of an arithmetic expression) is passed where a non-constant reference parameter is expected. There is also a warning when `x` is used before being assigned a value:

```
int main()
{
    int x;
    const int y = 2;

    value(1);
    value(x);                               // Warning, value used before set.
    value(y);
    value(x+1);

    reference(1);                           // Error. Non-const reference to const.
    reference(x);
    reference(y);                           // Error. Non-const reference to const.
    reference(x+1);                         // Error. Non-const reference to const.

    const_reference(1);
    const_reference(x);
    const_reference(y);
    const_reference(x+1);

    return 0;
}
```

[Prev] [Home] [Next]

POOMA Tutorials
Managing Threads Explicitly

The program shown in this appendix is the most complicated to appear in these tutorials. Building on tutorial 4, it sums the values in an array, taking the layout of that array into account. Unlike the multi-patch and layout examples of tutorial 4, however, this program explicitly spawns threads to perform the accumulation on each patch of the array.

Much of this code should seem familiar---the specialized `accumulateWithLoop()` functions, for example, have already been discussed. The novelty lies in the classes `ResultHolder` and `ArrayAccumulator`, the templated function `spawn_accumulate()`, and the specialized accumulation function `accumulate()`. These are all discussed briefly after the code (which is in `examples/Patches/Threaded/Accumulate.h` in the release) is presented.

```
#ifndef ACCUMULATE_H
#define ACCUMULATE_H

#include <pthread.h>

template<int D, class T, class E> class ConstArray;
template<int D> class UniformGridLayout;

//-----------------------------------------------------------------------
// The guts of the accumulation algorithm.
// Specialized here for dimension 1, 2 and 3.
// Can't call these 'accumulate' because it would be ambiguous.
//-----------------------------------------------------------------------

template<class T, class E>
inline T accumulateWithLoop(
    const ConstArray<1, T, E> & x
){
    T sum = 0;
    int f0 = x.first(0);
    int l0 = x.last(0);
    for (int i0=f0;i0<=l0; ++i0)
        sum += x(i0);
    return sum;
}

template<class T, class E>
inline T accumulateWithLoop(
    const ConstArray<2, T, E> & x
){
    T sum = 0;
    int f0 = x.first(0);
    int f1 = x.first(1);
    int l0 = x.last(0);
    int l1 = x.last(1);
    for (int i1=f1; i1<=l1; ++i1)
    {
```

```
        for (int i0=f0;i0<=l0; ++i0)
        {
            sum += x(i0, i1);
        }
    }
    return sum;
}

template<class T, class E>
inline T accumulateWithLoop(
    const ConstArray<3, T, E> & x
){
    T sum = 0;
    int f0 = x.first(0);
    int f1 = x.first(1);
    int f2 = x.first(2);
    int l0 = x.last(0);
    int l1 = x.last(1);
    int l2 = x.last(2);
    for (int i2=f2; i2<=l2; ++i2)
    {
        for (int i1=f1; i1<=l1; ++i1)
        {
            for (int i0=f0;i0<=l0; ++i0)
            {
                sum += x(i0, i1);
            }
        }
    }
    return sum;
}

//-----------------------------------------------------------------------
// The user interface for accumulate.
// Bricks just call the dimension specialized versions.
//-----------------------------------------------------------------------

template<int D, class T>
T accumulate(
    const ConstArray<D, T, Brick> & x
){
    return accumulateWithLoop(x);
}

template<int D1, class T, int D2, bool S>
T accumulate(
    const ConstArray<D1, T, BrickView<D2, S>> & x
){
    return accumulateWithLoop(x);
}

//-----------------------------------------------------------------------
// class ResultHolder<T>
//
// A class which holds the result of a calculation in such
// a way that you don't have to worry about how it got it.
// That is handled in subclasses.
//-----------------------------------------------------------------------
```

```
template<class T>
class ResultHolder
{
  public:
    ResultHolder()
    {}

    virtual ~ResultHolder()
    {}

    const T& get()
    {
        return result;
    }

  protected:
    T result;
};

//-----------------------------------------------------------------------
// class ArrayAccumulator<T, ArrayType>
//
// A specific type of calculation that returns using a ResultHolder.
// This holds an array of arbitrary type and accumulates the sum
// into the result.
//-----------------------------------------------------------------------

template<class T, class ArrayType>
class ArrayAccumulator : public ResultHolder<T>
{
  public:
    // Remember my type.
    typedef ArrayAccumulator<T, ArrayType> This_t;

    // Let the member data destroy itself.
    virtual ~ArrayAccumulator()
    {}

    // A static function that will be run in a thread.
    // The data passed in is an object of type This_t.
    static void *threadAccumulate(
        void * x
    ){
        This_t *y = static_cast<This_t*>(x);
        y->result = accumulate(y->array);
        return x;
    }

    // Construct with a const ref to an array.
    // Just remember the array.
    ArrayAccumulator(
        const ArrayType & a
    ) : array(a)
    {}

  private:
    // Store the array by value since the one passed in could be
    // a temporary.
    ArrayType array;
```

```
};

//-----------------------------------------------------------------------
// void spawn_thread(pthread_id, ArrayType)
//
// Spawns a thread that runs an ArrayAccumultor.
//-----------------------------------------------------------------------

template<class ArrayType>
inline void
spawn_accumulate(
    pthread_t &       id,
    const ArrayType & a
){
  // Typedefs to make the thread create more clear.
  typedef typename ArrayType::Element_t T;
  typedef ArrayAccumulator<T, ArrayType> Accumulator_t;

  // Spawn a thread:
  //   Store the id through the reference that is passed in.
  //   The function to call is threadAccumulate
  //   The thread data is an ArrayAccumulator using the passed in array.
  pthread_create(&id, NULL, Accumulator_t::threadAccumulate,
                 new Accumulator_t(a));
}

//-----------------------------------------------------------------------
// Multipatch version.
// Loop over patches and accumulate each patch.
//-----------------------------------------------------------------------

template<int D, class T>
T accumulate(
    const ConstArray<D, T, MultiPatch<UniformTag,Brick>> & x
){
    // Get the GridLayout from the array.
    const GridLayout<2>& layout = x.message(GetGridLayoutTag<2>());

    // Find the number of patches.  We'll have one thread per patch.
    int patches = layout.size();

    // An array of thread ids.
    pthread_t *ids = new pthread_t[patches];

    // Loop over patches.
    typename GridLayout<2>::iterator i= x.message(GetGridLayoutTag<2>()).begin();
    typename GridLayout<2>::iterator e= x.message(GetGridLayoutTag<2>()).end();
    int c=0;
    while (i!=e)
    {
        // Spawn a thread for each patch.
        // cout << "spawn" << endl;
        spawn_accumulate(ids[c], x(*i));
        ++i;
        ++c;
    }

    // Wait for all the threads to finish.
    // Get the sum from each, and accumulate that
```

```
        // in this thread.
        T sum = 0;
        for (int j=0; j<c; ++j)
        {
            // Wait for a given thread to finish.
            // cout << "join" << endl;
            void * v;
            pthread_join(ids[j], &v);

            // Get the result of the sum for that thread.
            // We don't need to know the array type for this.
            ResultHolder<T>* s = static_cast<ResultHolder<T>*>(v);
            cout << s->get() << endl;
            sum += s->get();

            // Delete the data structure passed to the thread.
            delete s;
        }

        // Return the full sum.
        return sum;
    }

    //-----------------------------------------------------------------------
    // General engine version.
    // If we don't know anything about the engine, at least get the right answer.
    //-----------------------------------------------------------------------

    template<int D, class T, class E>
    T accumulate(
        const ConstArray<D, T, E> & x
    ){
        return accumulateWithLoop(x);
    }

    #endif
```

We will not explain this code in detail, but rather will try to give an overview of the main issues it raises and addresses. First, the pthreads library requires programs to pass a `void*` data pointer when creating a thread, but the thing you pass to the other subroutines is a temporary (in this case, `x(*i)`). The program must therefore build an object (in this case an `ArrayAccumulator`) to store the array by value. While this must be built on the heap, not the stack, the `Array` object is still of course just a handle on the real data.

Second, since the program constructs an object to pass to the thread, it must destroy that object appropriately. In this case `pthread_join()` returns (via an argument) the pointer that was passed to it; the main `accumulate()` function picks up this pointer, and deletes the object it points to after casting it appropriately.

There is always the question of how the thread will return information to the rest of the code. In this case, since it is passing the `ArrayAccumulator` back through `pthread_join`, the `ArrayAccumulator` has the result of the sum for that thread.

`ArrayAccumulator` needs to know the exact type of `x(*i)` in order to do the accumulation, but it would be bad practice to make the subroutine that loops over the patches only work for one type of array. Instead, the program uses a function called `spawn_accumulate()`, which is templated on the actual array type.

The program has now handled the problem of generating the threads without knowing the type of `x(*i)`, but it still needs to receive the `ArrayAccumulator`, and that also has the type. The return data of type `T` is therefore split into the base class `ResultHolder`, which only knows the type `T`. The thing passed back from `pthread_join` is a pointer to that; since its destructor is virtual, it can safely be deleted.

The result is verbose, but not any more so than most multi-threaded programs. The biggest complication is having to introduce the `ArrayAccumulator` class in order to put the array being summed over on the heap instead of the stack.

[Prev] [Home] [Next]

POOMA Tutorials
Recommended Reading

Most computer bookstores have several shelves full of introductory books on C++. C++ for Fortran Programmers, by Ira Pohl, is among the better of these. The book is well organized, and covers all of the language's most useful features without becoming bogged down in details.

After working through one of those, everyone who plans to use the language should read Effective C++ by Scott Meyers, and Algorithms, Data Structures, and Problem Solving with C++ by Mark Weiss. Effective C++ (and its companion, More Effective C++) present dozens of guidelines on how to use C++ effectively. Always making destructors virtual, for example, makes it safer and easier to create heterogeneous collections of objects, while explicitly providing a copy constructor can prevent many hard-to-find aliasing bugs.

Weiss's book on data structures is a conventional textbook, but better written and more up-to-date than most. The author covers basic structures such as arrays, stacks, and queues before moving on to trees, hash tables, skip lists, and their more complicated kin. His presentation and analysis are concise and to-the-point, and the book provides complete implementations of all of the data structures it describes.

Almost all programming books talk about design; John Lakos's Large-Scale C++ Software Design is one of the few devoted to the problems that arise in actually implementing large programs. The book discusses ways to (re-)organize source code to reduce compilation time (from several days to overnight in one case), ease maintenance, and facilitate re-use.

Musser and Saini's STL Tutorial and Reference Guide is exactly what its title implies. The first part of the book explains what the C++ Standard Template Library (STL) is trying to accomplish; the middle introduces the STL's major features, and shows how they are used, while the back of the book is a reference guide.

Austern's Generic Programming and STL Book provides an excellent introduction to generic programming by introducing the notions of *concepts* and *models*. According to Austern, "a concept describes a set of requirements on a type, and when a specific type satisfies all of those requirements, we say that it is a model of that concept." A concept is not a C++ class, function, or template; however, any of these entities can serve as a model of a concept. Using these ideas, Austern also provides a complete reference for the STL.

Finally, see the POOMA web site for on-line presentations and technical papers describing the POOMA framework.

Bibliography

John Lakos: Large-Scale C++ Software Design. Addison-Wesley, 1997, ISBN 0201633620.

Ira Pohl: C++ for Fortran Programmers. Addison-Wesley, 1997, ISBN 0201924838.

Scott Meyers: Effective C++ (2nd ed.). Addison-Wesley, 1997, ISBN 0201924889.

Scott Meyers: More Effective C++. Addison-Wesley, 1995, ISBN 020163371X.

David R. Musser and Atul Saini: STL Tutorial and Reference Guide. Addison-Wesley, 1996, ISBN 0201633981.

Matthew H. Austern: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley, 1998, ISBN 0201309564.

Mark Weiss: Algorithms, Data Structures, and Problem Solving C++. Addison-Wesley, 1996, ISBN 0805316663.

[Prev] [Home] [Next]

POOMA Tutorials
Legal Notice

This software and ancillary information (herein called "SOFTWARE") called POOMA (Parallel Object-Oriented Methods and Applications) is made available under the terms described here. The SOFTWARE has been approved for release with associated LA-CC Number *LA-CC-98-65*.

Unless otherwise indicated, this SOFTWARE has been authored by an employee or employees of the University of California, operator of the Los Alamos National Laboratory under Contract No. W-7405-ENG-36 with the U.S. Department of Energy. The U.S. Government has rights to use, reproduce, and distribute this SOFTWARE, and to allow others to do so. The public may copy and use this SOFTWARE, FOR NONCOMMERCIAL USE ONLY, without charge, provided that this Notice and any statement of authorship are reproduced on all copies. Neither the Government nor the University makes any warranty, express or implied, or assumes any liability or responsibility for the use of this SOFTWARE.

If SOFTWARE is modified to produce derivative works, such modified SOFTWARE should be clearly marked, so as not to confuse it with the version available from LANL.

For more information about POOMA, send e-mail to pooma@acl.lanl.gov, or visit the POOMA web page at http://www.acl.lanl.gov/pooma.

[Prev] [Home]